# Una Introducción a la Teoría de la Computabilidad desde una perspectiva de Programación Lógica


TRABAJO DE FIN DE GRADO DEL DOBLE GRADO EN INGENIERÍA INFORMÁTICA - MATEMÁTICAS

**Clara Rodríguez Núñez**

**Director: Francisco Javier López Fraguas**
**Facultad de Informática**
**Universidad Complutense de Madrid**

**Mayo 2019**

Documento maquetado con TEXIS v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

# Una Introducción a la Teoría de la Computabilidad desde una perspectiva de Programación Lógica

*Memoria que se presenta para optar al título de Graduada en Ingeniería Informática y Matemáticas*

**Clara Rodríguez Núñez**

*Dirigida por el Profesor*

**Francisco Javier López Fraguas**

**Director: Francisco Javier López Fraguas**
**Facultad de Informática**
**Universidad Complutense de Madrid**

**Mayo 2019**

*A mi familia,*
*por acompañarme a buscar*
*dinosaurios cada domingo*

# Agradecimientos

En primer lugar, a Paco. Gracias por la oportunidad de hacer este trabajo contigo. Gracias por proponerme este tema del que no sabía nada y que ahora me encanta, gracias por leer todas mis demostraciones por absurdas o pesadas que fueran y por soportar que no sea capaz de saber cuándo como lleva tilde. De verdad, ha sido un auténtico privilegio tenerte como tutor en este trabajo. Gracias por todo.

Gracias a todos.

A mis padres. Por las cajas y cajas de dinosaurios y soportar las grandes marchas hasta mi cuarto. Por aprender a conducir para llevarme a un entrenamiento y cada tarde venir a las canchas. Por decirme que me fuera a un campamento a Rusia. Por hacerme creer que yo podía estudiar esto y aguantar mis crisis pre-exámenes. Por el enorme esfuerzo que supone mantener a una hija fuera de casa. Por escuchar a Los Lumineers antes de cada partido y aguantarme cuando vamos tarde. Por no daros cuenta de cuando me rompí la nariz.

A Miguel, por ser el mayor fan de Chuck Norris, Lebron James y mi mano izquierda. No sé cuál de esas tres cosas habla peor de tu gusto.

A mi abuela, gracias por ser la persona con mayor pasión por aprender que he conocido nunca. Gracias por todas las llamadas de los miércoles en las que me contabas algún dato nuevo sobre mi amiga la B12. Gracias por mantener a Siri a raya.

A Fernan, Carol, Rubén y Paula por todos los domingos que me habéis invitado a comer y por hacerme sentir que Madrid es también mi casa.

A los profesores y profesoras que he tenido estos años. Gracias por haberme transmitido vuestro amor por las matemáticas y la informática, por motivarme en vuestras clases a aprender y a aprovechar esta oportunidad al máximo. Tengo claro que si volviera 5 años atrás en el tiempo volvería a escribir esta carrera como mi primera opción. Gracias por ser los grandes responsables de ello.

Gracias al instituto público IES Arca Real. Gracias a todos los profesores y profesoras que día a día luchan por transmitir sus ganas de aprender. Si hoy yo puedo hacer este trabajo es por personas como vosotras, gracias por crear una educación pública donde todos tengamos cabida.

A mis compañeros de clase durante estos 5 años, especialmente a los que me han tenido que soportar también fuera de ella. A Álvaro Rodríguez, aunque nos abandones por ser gran "cientefico". A Belén, porque el 20 de septiembre es nuestro aniversario. A Ivan, por arreglar redes de camino al metro. A David, aunque no sepas hacer aguadillas. A Majo, aunque tú sí sepas hacerlas. A Miguel, por soportarme este año. A Álvaro García, por ser el creador del francoin.

A Laura y Lucía, por ser posiblemente el mejor regalo que me ha dado la infancia.

Al baloncesto. Gracias por darme un lugar donde sentirme completamente yo, aunque eso implique caerme sola de vez en cuando.

Gracias a las compañeras y entrenadores que he tenido esta temporada. Gracias por haberme hecho hecho reír más que nunca en una cancha. Por soportar mis intentos de explicación acerca de este trabajo y no asesinarme cada vez que celebraba un triple. Gracias a vosotras soy influencer.

Gracias a Rocío, por ser mi compañera fiel en esta aventura. En el Eco, China o la sala vip de algún hotel, pero siempre en mi equipo.

Gracias a Prats, por prestarme Eurovips y soportarme tantísimas cenas y meriendas, incluso cuando me da por robar comida.

Espero que con poneros aquí me haya ganado que algún día me invitéis al Sumo.

A Reg, por dejarme jugar con ella al trivolei.

A The Lumineers, sin ellos hacer este trabajo no habría sido ni la mitad de divertido.

Por último, gracias a Loren. Porque hace muchos años me dijiste que se me daban bien las matemáticas.

# Resumen

El objetivo de este trabajo fue desarrollar un modelo de computabilidad basado en programación lógica pura. Se desarroló un modelo Turing-completo basado en este tipo de programación que nos permitió demostrar algunos resultados clásicos de la Teoría de la Computabilidad.

Para ello se establecieron las distintas semánticas de los programas lógicos que se han tomado a lo largo de este trabajo. Se definió el concepto de función computable bajo esta nueva perspectiva y se demostró que se trata de un modelo de cómputo Turing-completo. Una vez probado esto, se desarrollaron resultados clásicos de la Teoría de la Computabilidad bajo esta nueva perspectiva. De este modo, se demostró la existencia de programa lógico universal desarrollando dicho programa y probando formalmente su corrección. Gracias a él fuimos capaces de demostrar la existencia de conjuntos no recursivos y no recursivamente enumerables en el ámbito de la programación lógica, que fue uno de los resultados clave de este trabajo. También se desarrollaron mediante técnicas de reducción numerosos ejemplos de conjuntos no recursivos. Finalmente, se demostraron algunos teoremas fundamentales de la Teoría clásica de la Computabilidad como son el Teorema de Rice, el Teorema s-m-n, el Teorema de Recursión y el Teorema del Punto Fijo. Se adaptó el enunciado de estos teoremas al modelo de computabilidad basado en programación lógica y se probaron bajo esta nueva perspectiva.

**Palabras clave:**
Computabilidad. Programación Lógica. Turing-completitud. Programa Universal. Conjunto recursivo. Teorema de Rice. Teorema s-m-n. Teorema de Recursión. Teorema del Punto Fijo.

# Abstract

The goal of this project was to develop a computation model based on logic programming. We developed a Turing complete model based on this type of programs that allowed us to prove some of the classic results of Computability Theory.

We established some different semantics for the logic programming and defined the concept of computable function under this new perspective. According to that definitions, we proved that this model of computation is Turing-complete. Once that had been proved, some important results of the Computability Theory were developed. In that way, the existence of a universal program for logic programming was proved by developing a universal program and formally proving its correctness. Because of it, we were able to prove the existence of non-recurssive and non-recursively enumerable sets under the logic programming paradigm, which was one of the most important results of this project. We used reductions in order to find more examples of non-recursive sets. Finally, we proved some important theorems of the Theory of Computability such as the Rice Theorem, the s-m-n Theorem, the Recursion Theorem and the Fixed Point Theorem. We adapted the statement of these theorems to our model of computation based on logic programming and proved them under this perspective.

**Keywords:**
Computability. Logic Programming. Turing complete. Universal Program. Recursive set. Rice Theorem. S-m-n Theorem. Recursion Theorem. Fixed Point Theorem.

# Índice

# Capítulo 1

# Introducción

Estudiar qué funciones van a poder ser computadas por un algoritmo es una de las principales cuestiones que la Teoría de la Computabilidad ha tratado de responder, estableciendo a lo largo de su desarrollo distintas definiciones del concepto de función computable.

La primera noción de función computable fue desarrollada por Alonzo Church y se conoce como $\lambda$-cálculo, (Church, 1936). Se trata de un modelo que nos permite expresar funciones, pero que no establece ningún mecanismo para calcularlas. Aun así, nos permite dar una primera definición de función computable: aquella que puede ser expresada como una $\lambda$-expresión. De todos modos, este modelo presenta el problema de no contar con ninguna clase de método de cómputo.

El matemático Alan Turing desarrolló un primer modelo de computabilidad basado en un método de cómputo. Este modelo está basado en la conocida como máquina de Turing, (Turing, 1937). De este modo, se establece la noción de función Turing-computable, que se refiere a toda función para la que existe una máquina de Turing que la computa. Turing demuestra que una función es Turing-computable si y solo si puede expresarse como una $\lambda-$expresión, por lo que ambas definiciones de función computable son equivalentes.

Esta equivalencia sirvió como motivación para desarrollar la conocida como Tesis de Church-Turing, que establece que una función puede ser calculada a través de un método efectivo de cálculo si y solo si es Turing-computable. Esta tesis es indemostrable formalmente, aunque suele asumirse como cierta.

Desde entonces se han desarrollado otros numerosos modelos de computabilidad variando el modelo de cómputo en que se basan. Algunos de los principales son las máquinas de registros, los While programas o las máquinas de Post-Turing, siendo todos ellos Turing-equivalentes y contribuyendo a reforzar así la Tesis de Church-Turing; para una compilación de un buen número de modelos de computabilidad, ver (Cutland, 1980) o (Fernandez, 2009).

Nos gustaría resaltar una noción de computabilidad desarrollada por Stephen Kleene, que definió el concepto de función recursiva parcial, (Kleene, 1936). Las funciones recursivas parciales son el menor conjunto de funciones parciales entre naturales que incluye las funciones cero, sucesor y proyección y que es cerrado respecto a la sustitución, recursión primitiva y minimización de funciones, como aparece desarrollado en (George S. Boolos, 2002). El carácter formal de esta definición, que no establece ningún modelo de cómputo para este tipo de funciones, nos será útil en el desarrollo de este trabajo, por lo que le prestaremos especial atención. Se puede demostrar que toda función es recursiva parcial si y solo si es Turing-computable, por lo que estas dos definiciones resultan equivalentes, como podemos encontrar en (Bridges, 1994).

El objetivo de este trabajo es desarrollar un modelo de computabilidad basado en Programación Lógica, teniendo como referencia (Leon S. Sterling, 1994). Hemos este elegido este paradigma de programación debido al carácter abstracto de su semántica y a la facilidad que suelen presentar los lenguajes

lógicos, como Prolog, para construir metaintérpretes, que constituyen esencialmente programas universales. Además, el carácter lógico de sus cláusulas, entendidas como implicaciones, nos permitirá razonar sobre ellos de manera sencilla.

Trataremos, de este modo, de desarrollar algunos de los principales resultados de la Teoría de la Computabilidad clásica sobre esta perspectiva de Programación Lógica. Creemos que puede ser interesante observar si algunos de los problemas clásicos de la Teoría de la Computabilidad van a simplificarse, como ocurrirá especialmente con el Teorema s-m-n y el desarrollo de un programa universal.

Dedicaremos el segundo capítulo de este trabajo a establecer la sintaxis y semántica que tomaremos a lo largo del trabajo para los programas lógicos, a los que llamaremos habitualmente PL-programas. Aunque este capítulo tenga un carácter introductorio, en él se definen conceptos que van a resultar clave a lo largo del trabajo.

En el tercer capítulo definiremos el concepto de función PL-computable y demostraremos que toda función recursiva parcial es PL-computable. Por lo tanto, el modelo que estamos desarrollando será Turing-completo. Este es uno de los resultados básicos de este trabajo ya que nos garantiza que el modelo con el que estamos trabajando podrá computar, al menos, las mismas funciones que una máquina de Turing.

El cuarto capítulo tendrá como objetivo desarrollar un PL-programa universal. Comprobaremos que este programa va a ser más sencillo que los desarrollados para otros modelos de cómputo, siendo capaces de demostrar además con precisión en este caso que efectivamente se comporta como un programa universal, algo que resulta prácticamente imposible para otros modelos como las máquinas de Turing o las máquinas de registros.

El quinto y sexto capítulo los dedicaremos a desarrollar los conceptos de conjunto PL-recursivo y PL-recursivamente enumerable. Encontraremos en ellos numerosos ejemplos de conjuntos no PL-recursivos a través de reducciones, lo que nos permitirá demostrar que existen funciones que los programas lógicos no son capaces de computar.

El séptimo capítulo lo dedicaremos a demostrar algunos de los principales teoremas de la Teoría de la Computabilidad. De esta manera, demostraremos el Teorema de Rice, el Teorema s-m-n, el Teorema de Recursión y el Teorema del Punto Fijo.

Finalizaremos el trabajo con un capítulo que incluirá las conclusiones a las que hemos llegado a través de este trabajo.

# Capítulo 2

# Sintaxis y Semántica PL

Vamos a dedicar este capítulo a establecer de forma clara la sintaxis y la semántica de nuestros programas lógicos. Vamos a adoptar una sintaxis muy similar a Prolog, ya que es el lenguaje de programación lógico más extendido y resulta de fácil comprensión.

Sin embargo, no vamos a hacer uso de las funciones metalógicas presentes en Prolog, utilizando únicamente el conocido como Prolog puro en el que todas las cláusulas de un programa deben entenderse sin más como fórmulas de la lógica de primer orden (cláusulas de Horn, un tipo de implicaciones cuantificadas universalmente, más exactamente). Quedan fuera, por tanto, recursos metalógicos o extra lógicos de Prolog, como son los cortes, negación, modificar la base de datos, etc. Esto se debe a que, como vamos a demostrar más adelante, aunque no tengamos dichas funciones en nuestros programas lógicos, estos continúan siendo Turing-completos, en el sentido de que vamos a poder computar con ellos todos los modelos alternativos previos, equivalentes a las máquinas de Turing.

La primera parte de este capítulo puede considerarse como un previo a lo que va a ser nuestro trabajo, limitándose a definir la sintaxis que tomaremos. Estableceremos la sintaxis para referirnos a los términos o cláusulas de un programa lógico y definiremos conceptos que resultarán clave para después desarrollar su semántica.

La segunda sección del capítulo presenta un mayor interés. En ella vamos a describir la semántica que vamos a dar a los programas lógicos. Presentaremos dos semánticas estándar en los programas lógicos: una basada en modelos mínimos y otra en puntos fijos del llamado operador de consecuencias inmediatas $T_p$, que definiremos más adelante. Acabaremos el capítulo con el bien conocido resultado de que las dos semánticas son equivalentes, por lo que en lo que resta de trabajo podremos pasar de la una a la otra sin correr ningún riesgo.

Tanto para definir la sintaxis como en la semántica nos hemos basado en (Leon S. Sterling, 1994, cap. 1), un texto de programación lógica clásico. En cambio, las demostraciones desarrolladas a lo largo del capítulo son propias.

## 2.1. Sintaxis PL

Vamos a definir en esta sección las construcciones y conceptos que vamos a manejar en nuestros programas lógicos.

Nuestros programas van a utilizar constructoras y variables. Por este motivo, antes de pasar a definir la sintaxis de nuestros programas vamos a definir de manera abstracta los conjuntos de los que tomaremos dichas variables y constructoras. Esto nos va a permitir definir la sintaxis de nuestros PL-programas desde un alto nivel de abstracción.

Comenzaremos por considerar el conjunto que va a englobar las constructoras que manejarán nues-

tros programas, se trata de la signatura de constructoras.

**Definición 2.1.1** (Signatura de constructoras)**.** *Una signatura de constructoras es un conjunto de símbolos $C$ de modo que cada $c \in C$ tiene asociada una aridad $ar(c) \in \mathbb{N}$, que expresa el número de argumentos al que está previsto que se aplique el símbolo $c$.*

*Escribiremos a veces $c/n$ para indicar que $c \in C$ y su aridad es $n$. Cuando $n = 0$, decimos que $c$ es una costante.*

Las signaturas que consideraremos son numerables, y en ocasiones pediremos que sean finitas.

Por otro lado, asumimos también un conjunto numerable $V$ de variables. Para distinguir notacionalmente constructoras y variables concretas, comenzaremos las variables por mayúsculas $(X, Y, \dots)$ y las constructoras por minúsculas $(cero, suc, a, b, \dots)$.

A partir de la signatura de constructoras y el conjunto de variables vamos a poder definir los términos de nuestros programas lógicos.

**Definición 2.1.2** (Término)**.** *El conjunto de términos correspondiente a $C$ y $V$, que notaremos por $\mathcal{T}_C(V)$, es el menor conjunto que verifica:*

1. $x \in \mathcal{T}_C(V)$, $\forall x \in V$.
2. $c \in \mathcal{T}_C(V)$, *si* $c/0 \in C$.
3. $c(t_1, \dots, t_n) \in \mathcal{T}_C(V)$, *si* $c/n \in C$, $t_1, \dots, t_n \in \mathcal{T}_C(V)$.

Vamos a establecer ahora una distinción entre nuestros términos, que será fundamental de cara a establecer la semántica de nuestros programas.

**Definición 2.1.3** (Término básico)**.** *Decimos que un término $t \in \mathcal{T}_C(V)$ es básico (ground) si en $t$ no hay apariciones de variables.*

*Notamos por $\mathcal{T}_C$ al conjunto de términos básicos correspondientes a $C$.*

De este modo, si tomamos $C = \{cero/0, suc/n\}$ y $V = \{X\}$, denotando por $suc^i(x)$ a la constructora $suc$ aplicada $i$ veces sobre $x$, entonces $\mathcal{T}_C(V) = \{suc^i(cero), suc^i(X) : i = 0, 1, \dots\}$. Por otro lado, $\mathcal{T}_C = \{suc^i(cero) : i = 0, 1, \dots\}$.

En la siguiente sección vamos a ver que existe una gran diferencia en el significado de los términos según sean o no básicos.

Pasamos ahora a definir algunos de los conceptos clave sobre nuestra sintaxis, que van a tener mucha importancia de cara a posteriormente definir la semántica de nuestros Programas Lógicos. Se trata de las sustituciones y las instancias.

**Definición 2.1.4** (Sustitución)**.** *Decimos que una sustitución es un conjunto finito de pares de la forma $X = t$ donde $X$ es una variable y $t$ es un término, donde ninguna variable que aparezca en el lado izquierdo de un par aparece en el lado derecho de ningún par ni dos pares distintos tienen la misma variable en el lado izquierdo. En otros textos se llaman sustituciones idempotentes, pero son las únicas que aquí consideramos.*

*Para cualquier sustitución $\omega = \{X_1 = t_1, \dots, X_n = t_n\}$ y un término $s$, denotaremos por $s\omega$ al resultado de sustituir simultáneamente cada aparición de la variable $X_i$ por $t_i$ para $i$ de 1 a $n$.*

**Definición 2.1.5** (Instancia)**.** *Decimos que un término $t$ es instancia de otro término $s$ si existe una sustitución $\omega$ tal que $t = s\omega$.*

Ahora que ya hemos dejado claro cuál es la sintaxis de nuestros términos, así como descrito sus principales propiedades, pasamos a definir el elemento que va a conformar nuestros programas lógicos.

**Definición 2.1.6** (Claúsula). *Una claúsula (o regla) es de la forma:*

$A$ :- $B_1, B_2, \ldots, B_k$.

*con $k >= 0$, donde $A$ y los $B_i$ son términos que no son variables.*

*Llamaremos objetivos atómicos a cada uno de los términos $A, B_1, \ldots, B_k$. Llamaremos a $A$ cabeza de la cláusula y a la secuencia $B_1, \ldots, B_k$ cuerpo. Si $k = 0$ la cláusula suele conocerse como "hecho" y escribirse como $A$.*

Habitualmente se suele distinguir entre los nombres de predicado, utilizados para referirse al símbolo más externo de un objetivo atómico, y nombre de constructora, y por tanto, entre término y objetivo atómico. De este modo, se establecerían dos signaturas de símbolos distintas: una relativa a los términos y otra para los objetivos atómicos, que se correspondería con los predicados.

Sin embargo, en nuestro caso no vamos a hacer esta distinción. Vamos a preferir mantener la unidad sintáctica entre ambos, ya que esta unidad va a ser importante de cara a desarrollar un programa lógico universal como haremos más adelante. Por este motivo, solo hablaremos de variables y constructoras.

De todos modos, seguiremos utilizando la terminología de predicado para referirnos al símbolo de constructora utilizado como símbolo más externo aplicado a un objetivo atómico, aunque en nuestro caso los predicados formen parte de nuestra signatura de constructoras.

Ya solo nos queda por definir qué es un Programa Lógico.

**Definición 2.1.7** (Programa Lógico). *Un Programa Lógico es un conjunto finito de cláusulas.*

Hemos hablado de un conjunto finito de cláusulas porque es lo estándar, pero a efectos de este trabajo se podría hablar de multiconjunto o incluso de secuencias de cláusulas. Esto se debe a que el desarrollo que vamos a realizar va a basarse en una semántica de los programas en la que es irrelevante cual de estas nociones se haya elegido.

Incluimos un breve ejemplo de Programa Lógico de cara a dejar clara la sintaxis que hemos tomado. Para este ejemplo trabajamos con $C = \{cero/0, suc/1, par/1\}$ y $V = \{X\}$.

```
par(cero).
par(suc(suc(X))) :- par(X).
```

El objetivo de esta sección es únicamente fijar sintaxis que hemos tomado en nuestros Programas Lógicos. Aún no hemos definido su significado, que es lo que haremos en la siguiente sección estableciendo su semántica.

## 2.2. Semántica PL

Comenzamos dando una serie de definiciones que serán básicas a la hora de desarrollar nuestra teoría sobre Programas Lógicos. Partiendo de una signatura de constructoras cualquiera $C$ y un conjunto de variables $V$, trataremos de dar la semántica de los PL-programas definidos sobre ellas.

De cara a trabajar con programas lógicos se suelen definir los conceptos de Universo de Herbrand (formado por todos sus términos básicos) y Base de Herbrand de un programa (formado por los objetivos atómicos básicos). Sin embargo, nosotros no vamos a tener que realizar esta distinción ya que en

la sintaxis de los PL-programas no hemos distinguido entre objetivos y términos.

De este modo, trabajaremos sobre $\mathcal{T}_C$ para referirnos a los términos básicos sobre los que está definido nuestro programa. Dar un significado a nuestros programas lógicos consistirá en establecer qué elementos de $\mathcal{T}_C$ son los que forman parte de su semántica.

Para ello introducimos las siguientes definiciones:

**Definición 2.2.1** (Interpretación). *Una interpretación $I$ de $P$ es un subconjunto de $\mathcal{T}_C$, o lo que es lo mismo, $I \in \mathcal{P}(\mathcal{T}_C)$.*

Dentro las interpretaciones de un programa vamos a estar especialmente interesados en aquellas que son coherentes con las cláusulas de $P$ entendidas como fórmulas lógicas. Son las que conocemos como modelos y que pasamos ahora a estudiar. Nos van a permitir dar una semántica declarativa a nuestros programas lógicos.

**Definición 2.2.2** (Modelo). *Una interpretación $I$ de $P$ es modelo si para cada instancia básica de una cláusula del programa $A \leftarrow B_1, ..., B_n$, se tiene que si $B_1, ..., B_n \in I$ entonces $A \in I$.*
*Notamos por $Mod(P)$ al conjunto de todos los modelos de $P$.*

Veamos ahora un resultado interesante que nos va a permitir definir la existencia de modelo mínimo con relación al orden de inclusión $\subseteq$.

**Lema 2.2.1** (Intersección de modelos). *La intersección de una familia no vacía de modelos es un modelo.*

*Demostración.* Sean $\mathcal{F}$ familia no vacía de modelos de $P$. Veamos que $\bigcap \mathcal{F}$ también es modelo.

Sea $A \leftarrow B_1, ..., B_n$ una instancia básica a una cláusula de $P$, supongamos que $B_1, ..., B_n \in \bigcap \mathcal{F}$.

Para cada $M \in \mathcal{F}$ tenemos que como $B_1, ..., B_n \in M$ entonces, por ser $M$ modelo, se tiene que $A \in M$.

De este modo, $A \in M$ para cada $M \in \mathcal{F}$, por lo que $A \in \bigcap \mathcal{F}$

□

Ahora que disponemos de este resultado, pasamos a definir el modelo mínimo de nuestro programa, habiendo garantizado previamente que la intersección de una familia no vacía de modelos es un modelo.

Para ello aprovechamos el hecho de que todo PL-programa tiene siempre algún modelo, pues $\mathcal{T}_C$ es siempre modelo de $P$, con lo que podemos asegurar que $\bigcap_{M \in Mod(P)} M$ es modelo ya que $Mod(P) \neq \emptyset$.

**Definición 2.2.3** (Modelo mínimo). *Se conoce como modelo mínimo de un programa $P$ al modelo, que notamos por $M(P)$, obtenido como intersección de todos los modelos de nuestro programa. Se corresponde con la semántica declarativa de nuestro programa.*

*Es decir,* $M(P) = \bigcap_{M \in Mod(P)} M$

Como podemos ver, acabamos de dar un primer significado a $P$ al definir su semántica declarativa. Pasamos ahora a ver otra semántica de nuestros programas PL, la de punto fijo, que no solo los va a dotar de un significado sino que, a diferencia de la declarativa, nos va a dar un procedimiento para construirlo.

Por último, acabaremos la sección viendo que ambas semánticas son equivalentes en el sentido de que dan a nuestro programa el mismo significado.

### 2.2.1. Semántica de punto fijo

Dedicamos a definir en esta sección la semántica de punto fijo de nuestros PL-programas. Esta va a resultar de gran importancia dado que gran parte de las demostraciones que van a desarrollarse en este trabajo están basadas en ella.

El nombre de semántica de punto fijo se debe a que va a dar a nuestros PL-programas el valor del punto fijo de una función sobre ellos (más exactamente, sobre sus interpretaciones) que pasamos a definir ahora. Se va a tratar de una función continua sobre un ccpo (conjunto parcialmente ordenado completo por cadenas), lo que va a garantizar la existencia de tales puntos.

Comenzamos por lo tanto definiendo tal función para nuestro programa $P$. Aunque pueda parecer en un primer momento complicada, vamos a ver que su significado nos va a resultar mucho más natural:

**Definición 2.2.4** ($T_p$)**.** *Definimos la función $T_p$, denominada usualmente operador de consecuencias inmediatas, como $T_P : \mathcal{P}(\mathcal{T}_C) \longrightarrow \mathcal{P}(\mathcal{T}_C)$ tal que $T_P(I) = \{A \in \mathcal{T}_C : A \leftarrow B_1, \ldots, B_n, n \geq 0$, es una instancia básica de una cláusula en $P$ y $B_1, \ldots, B_n$ están en $I\}$.*

Para hacernos una idea del significado de esta función, vamos a intentar explicar de modo informal qué es lo que hace. Dada una interpretación $I$ lo que nos va a devolver $T_P(I)$ son todas las cabezas de instancias básicas de cláusulas de $P$ cuyo cuerpo está en $I$. Es decir, todos los hechos básicos que se deducen como ciertos a partir de los hechos de $I$ y de las cláusulas de $P$ entendidas como implicaciones lógicas.

Pasamos a probar rápidamente una serie de propiedades sobre esta función que nos van a permitir asegurar la existencia de puntos fijos en ella. Para las nociones de continuidad y monotonía nos referiremos a las utilizadas en (Hanne Riis Nielson, 2007, cap. 5) en ccpos. Tomamos como orden entre interpretaciones la inclusión conjuntista como mencionamos anteriormente.

**Lema 2.2.2** (Monotonía de $T_p$)**.** *La función $T_p$ es monótona. Esto es, si $I \subseteq J$ entonces $T_P(I) \subseteq T_P(J)$.*

*Demostración.* Si $A \in T_P(I)$ entonces existe $A \leftarrow B_1, \ldots, B_n$ instancia básica de una cláusula de $P$ con $B_1, \ldots, B_n \in I$.

Entonces, como $I \subseteq J$, tenemos que la misma $A \leftarrow B_1, \ldots, B_n$ es una instancia básica de una cláusula de $P$ con $B_1, \ldots, B_n \in J$ con lo que concluimos por la definición de $T_p$ que $A \in T_P(J)$. □

**Lema 2.2.3** (Continuidad de $T_p$)**.** *La función $T_p$ es continua.*

*Demostración.* Veamos que para toda cadena creciente de interpretaciones no vacía $Y$ se verifica que $\bigcup\{T_P(I) : I \in Y\} = T_P(\bigcup Y)$

1. Del hecho de que $T_P$ es monótona se desprende que $\bigcup\{T_P(I) : I \in Y\} \subseteq T_P(\bigcup Y)$.

2. Veamos ahora que $T_P(\bigcup Y) \subseteq \bigcup\{T_P(I) : I \in Y\}$. Si $A \in T_P(\bigcup Y)$ entonces tenemos, por definición de $T_P$, que existe $A \leftarrow B_1, \ldots, B_n$ instancia básica de una cláusula de $P$ cumpliendo que $B_1, \ldots, B_n \in \bigcup Y$.

   Supongamos ahora que $\nexists I \in Y$ tal que $B_1, \ldots, B_n \in I$ y veamos que llegamos a un absurdo. Para cada $B_i, i = 1 \ldots n$ tenemos que existe $I_i \in Y$ tal que $B_i \in I$. En caso contrario, podríamos tomar $\bigcup Y \setminus B_i$ y seguiría siendo cota superior de Y (absurdo). Ahora, como se trata de un número finito de cláusulas, tendremos que dadas nuestras $I_1, \ldots, I_n \in Y$ una de ellas (a la que denotaremos por $I'$) se corresponderá con $\bigcup I_1, \ldots, I_n$ y así contendrá a $B_1, \ldots, B_n$.

   Concluimos que como $B_1, \ldots, B_n \in I'$ entonces $A \in T_P(I')$. Dado que $I' \in Y$ se tiene que $A \in \bigcup\{T_P(I) : I \in Y\}$.

□

Una vez que hemos demostrado estas propiedades sobre $T_P$ pasamos ya a dar nuestra definición de la semántica de punto fijo de nuestros programas lógicos. Para ver que esta es correcta nos basamos en

el siguiente resultado sobre puntos fijos obtenido de (Hanne Riis Nielson, 2007), del cual no añadimos su demostración.

Dado un ccpo $D$ denotaremos por $\bot$ a su cota inferior, en la misma referencia podemos encontrar una demostración de su existencia. Por otro lado, nos referiremos por $f^i(x)$ a aplicar $i$ veces la función $f$ de manera consecutiva sobre el elemento $x$, es decir, a $\underbrace{f(f(\dots f}_{i}(x)\dots))$.

**Teorema 2.2.1** (Teorema del punto fijo). *Dado $D$ ccpo, tenemos que:*

1. *Si $f : D \to D$ es monótona, entonces existe $\bigcup\{f^n(\bot) : n \geq 0\} \in D$.*
2. *Si $f : D \to D$ es continua, tenemos que el mínimo punto fijo de $f$, al que denotamos por $FIX\ f$, existe y además, $FIX\ f = \bigcup\{f^n(\bot) : n \geq 0\} \in D$.*

Como podemos observar, nuestra función $T_P$ se encuentra en la situación descrita por el enunciado, con lo cual podemos trabajar con $FIX\ T_P$. Finalmente, pasamos a dar la definición de semántica de punto fijo de los PL-programas.

**Definición 2.2.5** (Semántica de punto fijo). *Dado un PL-programa $P$ definimos su semántica de punto fijo $M_{den}$ como*

$$M_{den}(P) = FIX\ T_P = \bigcup\{f^n(\emptyset) : n \geq 0\} \tag{2.1}$$

Para acabar esta sección, incluimos una demostración de la equivalencia de la semántica de punto fijo y la declarativa de los PL-programas.

**Lema 2.2.4** (Equivalencia semánticas declarativa y de punto fijo). *Dado un PL-programa $P$ se tiene que:*

$$M_{den}(P) = M(P) \tag{2.2}$$

*Demostración.* 1. Veamos en primer lugar que $FIX\ T_P$ es modelo.

Por ser punto fijo, tenemos que $FIX\ T_P = T_P(FIX\ T_P)$. Sea ahora $A \leftarrow B_1, ..., B_n$ una instancia básica de una cláusula de $P$, si $B_1, .., B_n \in FIX\ T_P$ entonces $A \in T_P(FIX\ T_P) = FIX\ T_P$.

De este modo, concluimos que $FIX\ T_P$ es modelo. Evidentemente,

$\bigcap_{\text{M modelo de P}} M \subseteq FIX\ T_P$.

2. Si $M$ modelo de $P$, veamos que $FIX\ T_P \subseteq M$.

En primer lugar, veamos que $T_P^n(\emptyset) \subseteq M$ para todo $n \geq 0$, lo probamos por inducción.

Para $n = 0$, resulta trivial que $\emptyset \subseteq M$.

Ahora, si para $n$ se verifica, veamos que también se verifica para $n + 1$.

Como $T_P^n(\emptyset) \subseteq M$ entonces por ser $T_P$ monótona tenemos que $T_P^{n+1}(\emptyset) \subseteq T_P(M)$.

Pero al ser $M$ modelo $T_P(M) \subseteq M$. Podemos comprobarlo observando que si $A \in T_P(M)$ entonces existe una instancia básica de una cláusula de $P$ tal que $A \leftarrow B_1, ..., B_n$ con $B_1, ..., B_n \in M$, y por ser $M$ modelo entonces $A \in M$.

De este modo, $T_P^n(\emptyset) \subseteq M$ para todo $n \geq 0$, con lo que $M$ será cota superior de la cadena que forman y así $FIX\ T_P \subseteq M$

Con estas dos implicaciones llegamos al resultado que buscábamos y podemos concluir que $M_{den}(P) = M(P)$. □

Una vez que ya hemos dejado definida la semántica de nuestros programas empezamos a desarrollar la idea de PL-computabilidad. En el próximo capítulo vamos a definir este concepto y estudiar la relación entre esta noción de computabilidad y la Turing-computabilidad.

# Capítulo 3

# PL-computabilidad

En este capítulo vamos a introducir el concepto básico que vamos a tratar de desarrollar en este trabajo: la PL-computabilidad.

El objetivo que perseguimos es trasladar la idea de que una función sea Turing-computable (es decir, que exista una máquina de Turing que la compute) a nuestra perspectiva de Programas Lógicos. Nuestro objetivo será que una función sea PL-computable si y solo si es Turing-computable.

Pero como explicaremos, no vamos a poder establecer esta correspondencia de manera directa. Las máquinas de Turing trabajan sobre naturales, mientras que nuestros PL-programas lo hacen sobre términos. Por este motivo, tendremos que adaptar esta equivalencia para que realmente sea cierta.

Comenzaremos este capítulo desarrollando el concepto de función PL-computable y explicando cuál es la relación que van a mantener con las funciones Turing-computables. Después pasaremos a desarrollar el resultado fundamental de este capítulo: la Turing-completitud de las funciones PL-computables.

Nos gustaría resaltar la importancia de este capítulo. No solo estamos definiendo qué es computar una función en el ámbito de los PL-programas (lo que será la base para todo el desarrollo que haremos) sino que estamos demostrando que estudiar los PL-programas tiene la misma potencia que estudiar las máquinas de Turing. Esta es la principal motivación para el desarrollo que haremos en lo que resta de trabajo, que tratará de trasladar resultados clásicos de la Teoría de la Computabilidad a la perspectiva de las funciones PL-computables.

## 3.1. Función PL-computable

Como hemos dicho, vamos a tratar de trasladar la idea de computar una función a nuestros Programas Lógicos.

Una primera diferencia que encontramos respecto a las funciones Turing-computables es que mientras que estas segundas están definidas sobre naturales, las funciones PL-computables estarán definidas sobre términos. Además, estos términos dependen de la signatura de constructoras que hayamos fijado.

Esto nos va a llevar a tener que afrontar unas pequeñas dificultades a la hora de establecer la relación entre estas dos nociones de computabilidad, pero hemos decidido hacerlo así para poder relacionar las funciones directamente con los PL-programas de la computan.

Pasamos, por tanto, a definir el concepto de función PL-computable.

**Definición 3.1.1** (Función PL-computable)**.** *Dada una signatura de constructoras $C$, $\mathcal{T}_C$ el conjunto de términos básicos formados con $C$, diremos que una función parcial $f : \mathcal{T}_C^n \hookrightarrow \mathcal{T}_C^m$ es PL-computable si podemos definir un PL-programa $P$ en el que podemos distinguir un predicado $f_P/n+m$ tal que $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$ si y solo si el término $f_P(x_1, \ldots, x_n, y_1 \ldots, y_m) \in M(P)$ para todo*

$x_1, \ldots, x_n, y_1, \ldots, y_m \in \mathcal{T}_C$ .

De cara a aclarar este concepto que acabamos de definir incluimos ahora un breve ejemplo de función PL-computable.

**Ejemplo 3.1.1.** *Tomamos la signatura de constructoras $C = \{cero/0, suc/1\}$. De cara a simplificar la notación nos referiremos por $suc^i(cero)$ al término obtenido al aplicar $i$ veces la constructora suc sobre la constante cero.*

*Vamos a probar que la función $f$ que para un término $t$ nos devuelve cero si se ha aplicado un número par de veces el constructor suc en el término (es decir, si $i$ es par) y $suc(cero)$ en caso contrario es PL-computable.*

*Tomamos $P$ el PL-programa formado por las claúsulas $R_1$, $R_2$ y $R_3$ siguientes:*

$R_1$: `f_P(cero, cero).`
$R_2$: `f_P(suc(X), cero) :- f_P(X, suc(cero)).`
$R_3$: `f_P(suc(X), suc(cero)) :- f_P(X, cero).`

*Veamos que computa la función $f$, para ello tenemos que probar que $f(suc^n(cero)) = T \Leftrightarrow f_P(suc^n(cero), T) \in M(P)$ para todo $n$. Lo demostramos por inducción.*

1. *Para $n = 0$, tenemos que de la única cláusula de cuya cabeza $f_P(cero, X)$ puede ser instancia es $R_1$. De este modo, $f_P(cero, z) \in M(P) \Leftrightarrow z = cero$, con lo que se verifica trivialmente la propiedad.*
2. *Estudiamos ahora el caso de $n+1$, asumiendo que se cumple en $n$. Distinguimos dos casos según $n+1$ sea impar o par:*

   a) *Si $n+1$ es par (y por lo tanto $n$ es impar), tenemos que $f(suc^{n+1}(cero)) = cero$. Veamos ahora que $f_P(suc^{n+1}(cero), z) \in M(P) \Leftrightarrow z = cero$.*

   *El predicado $f_P(suc^{n+1}(cero), z)$ puede ser instancia de las cabezas de las claúsulas $R_2$ y $R_3$, de manera que formará parte de la semántica si y solo si alguno de los cuerpos de estas cláusulas se verifica. Estudiamos como se comporta para cada una de ellas.*

   *Para el caso de $R_2$, tenemos que $f_P(suc^{n+1}(cero), cero) \in M(P) \Leftrightarrow f_P(suc^n(cero), suc(cero)) \in M(P)$. Está claro que $n$ es impar, con lo que por nuestra hipótesis de induccción como $f(suc^n(cero)) = suc(cero)$ entonces $f_P(suc^n(cero), suc(cero)) \in M(P)$. Volviendo a la anterior expresión, llegamos a que $f_P(suc^{n+1}(cero), cero) \in M(P)$.*

   *Para $R_3$, tenemos que $f_P(suc^{n+1}(cero), suc(cero)) \in M(P) \Leftrightarrow f_P(suc^n(cero), cero) \in M(P)$. Pero por hipótesis de inducción, si $f(suc^n(cero)) = suc(cero)$ entonces $f_P(suc^n(cero), cero) \notin M(P)$. De esta manera podemos concluir que $f_P(suc^{n+1}(cero), suc(cero)) \notin M(P)$.*

   *Hemos estudiado todas las cláusulas de las que $f_P(suc^{n+1}(cero), z)$ puede ser instancia y observamos que el único valor de $z$ para el que este predicado forma parte de la semántica es $z = cero$, como queríamos probar.*

   b) *El caso de $n+1$ par es análogo, por lo que no le incluimos.*

   *Hemos probado que $f(x) = z \Leftrightarrow f_P(x, z) \in M(P)$ para todo $x \in \mathcal{T}_C$, con lo que $f$ es una función PL-computable.*

Una vez que hemos definido qué es una función PL-computable y visto un ejemplo de ella, pasamos a estudiar ahora su relación con las funciones Turing-computables.

El resultado que vamos a tratar de demostrar es que toda función Turing-computable también va a ser PL-computable. Esta propiedad de un modelo de computabilidad (el ser capaz de computar todas las funciones Turing-computables) se conoce como Turing-completitud.

Se ha demotrado que numerosos modelos son Turing-completos. Ejemplos de ellos son las funciones recursivas parciales, las máquinas de registros o el Lambda cálculo, así como la mayoría de los lenguajes de programación. En estos casos se ha podido ir incluso más allá, demostrando que estos modelos no solo son Turing-completos sino que son Turing-equivalentes, es decir, que una función es Turing-computable si y solo si puede ser computada en estos modelos. De este modo, estos modelos refuerzan la tesis de Church-Turing que establece que una función sobre naturales puede calcularse a través de un método efectivo si y solo si es Turing-computable.

Tomaremos en adelante en este trabajo la siguiente nomenclatura: nos referiremos por $\mathcal{L}$ al conjunto de funciones PL-computables y por $\mathcal{T}$ al de las Turing-computables.

## 3.2. Turing completitud de las funciones PL-computables

Dedicaremos esta sección a probar que toda función Turing-computable va a tener una función equivalente PL-computable.

Una primera idea para idea para demostrar este resultado, que es la seguida en las demostraciones como la de (L.Hein, 2005), es simular una máquina de Turing Universal mediante un PL-programa. Nosotros hemos optado por hacer una demostración distinta.

Para ello vamos a manejar el conjunto de funciones recursivas parciales, que es Turing-equivalente como se demuestra en (Kleene, 2009). Estas funciones constituyen uno de los modelos de computabilidad más populares, que se caracteriza por un alto nivel de abstracción matemática desligada de ninguna idea de dispositivo o programa. Incluimos ahora la definición de función recursiva parciales, que hemos obtenido de (Cutland, 1980) :

**Definición 3.2.1** (Función recursiva parcial)**.** *El conjunto de funciones recursivas parciales es el conjunto más pequeño de funciones parciales de naturales en naturales de cualquier aridad que contiene las funciones básicas cero, siguiente y las proyecciones, tal que la composición, recursión y minimización son cerradas en él.*

Vamos a explicar brevemente cuáles son estas funciones y operaciones, así como la sintaxis que emplearemos para referirnos a ellas.

1. La función básica *zero* está definida como $zero(n) = 0$.
2. La función básica *siguiente* está definifa como $siguiente(n) = n + 1$.
3. La función $proy_n^i$ está definida como $proy_n^i(x_1, \ldots, x_i, \ldots, x_n) = x_i$.
4. La sustitución de las funciones $f_1(x_1, \ldots, x_n), \ldots, f_k(x_1, \ldots, x_n)$ en la función $g(x_1, \ldots, x_k)$ está definida como la función $h(x_1, \ldots, x_n) = g(f_1(x_1, \ldots, x_n), \ldots, f_k(x_1, \ldots, x_n))$.
5. La recursión de la función $f(x_1, \ldots, x_n, y, z)$ con la función $g(x_1, \ldots, x_n)$ para el caso base está definida como

   $h(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$

   $h(x_1, \ldots, x_n, n+1) = f(x_1, \ldots, x_n, n, h(x_1, \ldots, x_n))$
6. La minimización de una función $f(x_1, \ldots, x_n, y)$ está definida como la función $\mu f(x_1, \ldots, x_n) = n$ siendo $n$ el menor natural tal que

   *a)* $f(x_1, \ldots, x_n, y)$ está definido y es distinta de 0 para todo $y \leq n$.

   *b)* $f(x_1, \ldots, x_n, n) = 0$.

   si tal $n$ existe. Si no, $\mu f(x_1, \ldots, x_n)$ está indefindo.

Hemos decidido relacionar las funciones PL-computables con ellas porque con las funciones PL-computables nos encontramos en una situación similar: hemos dotado a nuestros programas de una semántica de carácter abstracto, sin ofrecer ningún mecanismo de cálculo.

Como las funciones recursivas parciales son Turing-equivalentes, si conseguimos probar que toda función recursiva parcial es PL-computable habríamos llegado a que el conjunto de las funciones PL-computables es Turing-completo.

Creemos que esta demostración resulta interesante, y hemos preferido a desarrollarla antes que la de construir el Turing-intérprete, porque va a manejar PL-programas muy sencillos sobre los que podremos trabajar formalmente sobre su semántica y así demostrar que su comportamiento es el deseado.

Seguiremos unas ideas similares a las que podemos encontrar en la demostración de la Turing-completitud de las funciones computables por máquinas de registros (Cutland, 1980), aunque ahora adaptando la demostración a los PL-programas.

Nos encontramos ahora con una primera dificultad a la hora de demostrar este resultado. Tenemos que, mientras que las funciones recursivas parciales trabajan sobre naturales, nuestros PL-programas están definidos sobre términos. Por este motivo no vamos a poder hablar de que una función recursiva parcial es igual a una PL-computable, sino que vamos a hablar de similares, siguiendo la noción de similitud que pasamos a desarrollar ahora.

Comenzamos definiendo la signatura de constructoras $\mathcal{N}$ sobre la que vamos a definir las funciones PL-computables en este caso. Será la formada por una constante a la que llamamos *cero* y el constructor de aridad 1 *suc*. El conjunto de términos que podremos formar a partir de ellos es $\mathcal{T}_{\mathcal{N}} = \{suc^n(cero) : n \geq 0\}$. Veamos como podemos establecer fácilmente una biyección entre él y $\mathbb{N}$.

$$\phi : \begin{array}{l} \mathbb{N} \to \mathcal{T}_{\mathcal{N}} \\ 0 \to cero \\ n+1 \to suc(\phi(n)) \end{array} \tag{3.1}$$

Una vez que hemos definido este conjunto pasamos a ver el resultado que nos ocupa. Como hemos dicho, no vamos a poder establecer la correspondencia directa, sino que tendremos que tener en cuenta la biyección que hemos construido entre $\mathbb{N}$ (donde están definidas las funciones recursivas parciales) y el conjunto $\mathcal{T}_{\mathcal{N}} = \{suc^n(cero) : n \geq 0\}$ (donde están definidas las PL-computables).

Tomaremos en adelante en este trabajo la siguiente nomenclatura: nos referiremos por $\mathcal{L}$ al conjunto de funciones PL-computables sobre la signatura $\mathcal{N}$ y por $\mathcal{T}$ al de las Turing-computables.

La prueba de este resultado es extensa al hacerla en detalle, pero resulta de mucho interés ya que, además de probar que toda función recursiva parcial puede computarse con PL-programas, nos da un procedimiento constructivo para generar el PL-programa que la compute.

**Teorema 3.2.1.** *Si $f(x_1, \ldots, x_n) \in \mathcal{T}$, entonces tenemos que existe $f' \in \mathcal{L}$ tal que*

$$f(x_1, \ldots, x_n) = y \Leftrightarrow f'(\phi(x_1), \ldots, \phi(x_n)) = \phi(y) \tag{3.2}$$

*-Obs: Diremos que toda función que verifique esta ecuación (independientemente de si es PL-computable o no) es similar a $f$.*

*Demostración.* Demostraremos que el conjunto de funciones recursivas parciales está contiene un conjunto similar al de las funciones PL-recursivas. Como $\mathcal{T}$ es el menor conjunto que contiene a las funciones *zero*, *siguiente* y las proyecciones y que es cerrado para sustituciones, recursiones y minimalizaciones, nos basta con demostrar que $\mathcal{L}$ también contiene funciones similares definidas sobre términos y es cerrado respecto a dichas operaciones.

Como hemos dicho, no vamos a poder referirnos a que contenga a la función recursiva parcial *zero* (o *siguiente* o $proy_n^i$), sino que trabajaremos con funciones similares definidas en $\mathcal{T}_\mathcal{N}$.

Pasamos a demostrar el resultado. Comenzamos viendo cómo las funciones recursivas parciales simples (la función *zero*,*sig* y $proy_n^i$) formarán parte de $\mathcal{L}$, para luego pasar a demostrar que también se verificará para los casos compuestos.

1. Para la función *zero*, tenemos que la función recursiva parcial *zero* está definida como $zero :: \mathbb{N} \to \mathbb{N} : zero(n) = 0, \forall n \in \mathbb{N}$.

   Demostraremos que la función $zero' :: \mathcal{T}_\mathcal{N} \to \mathcal{T}_\mathcal{N} : zero'(c) = cero, \forall c \in \mathcal{T}_\mathcal{N}$ (que es la función similar a *zero* definida sobre $\mathcal{T}_\mathcal{N}$) es PL-computable.

   Para ello tomamos el programa $P$ siguiente:

   `R: zero'`(X,cero).

   Estudiemos ahora cuándo $zero'_P(x, y) \in M(P)$ para $x \in \mathcal{T}_\mathcal{N}$. La única clásula de la que puede ser instancia este predicado es $R$, con lo que llegamos a que $zero'_P(x, y) \in M(P) \Leftrightarrow y = cero$ para todo $x \in \mathcal{T}_\mathcal{N}$.

   De este modo, tenemos que $P$ computará la función $zero'$.

2. Para la función *siguiente*, tenemos que $sig :: \mathbb{N} \to \mathbb{N} : sig(n) = n + 1, \forall n \in \mathbb{N}$ (utilizamos *sig* en vez de *siguiente* para simplificar la notación).

   De esta manera, buscamos demostrar que la función $sig' :: \mathcal{T}_\mathcal{N} \to \mathcal{T}_\mathcal{N} : sig'(c) = suc(c), \forall c \in \mathcal{T}_\mathcal{N}$ es PL-computable. Consideramos el siguiente PL-programa $P$:

   `R: sig'`(X,suc(X)).

   Con un razonamiento similar al del caso anterior llegamos a que $sig'_P(x, y) \in M(P)$ si y solo si $y = suc(x)$. De esta manera, queda claro que $P$ computa la función $sig'$, con lo que $sig'$ es PL-computable.

3. Por último, para el caso de las proyecciones, tenemos que la función $proy_i$ con $1 \leq i \leq n$ queda definida como $proy_i :: \mathbb{N}^n \to \mathbb{N} : proy_i(x_1, ..., x_n) = x_i$, para todo $x_1, ..., x_n \in \mathbb{N}$.

   Igual que antes, buscamos probar que es PL-computable la función $proy'_i :: \mathcal{T}_\mathcal{N}^n \to \mathcal{T}_\mathcal{N} : proy'_i(c_1, \ldots, c_n) = suc(c_i)$, para todo $c_1, \ldots, c_n \in \mathcal{T}_\mathcal{N}$.

   Tomamos el siguiente programa lógico $P$:

   `R: proy'`$_i$`(X`$_1$`,...,X`$_n$`,X`$_i$`).`

   Al igual que en los casos anteriores, tenemos que $proy'_P(x_1, \ldots, x_n, y)$ tan solo puede ser instancia de la cláusula $R$ con lo que se verifica para todos los $x_1, \ldots, x_n \in \mathcal{T}_\mathcal{N}$ que $proy'_P(x_1, \ldots, x_n, y) \in M(P)$ si y solo si $y = x_i$. De esta manera, $P$ computa la función $proy'_P$.

Con esto hemos demostrado que para cada función recursiva parcial básica podemos encontrar una función PL-computable similar. Veamos ahora que esto también ocurre en el caso de las funciones compuestas (formadas a través de sustituciones, recursiones o minimizaciones).

Vamos a comprobarlo caso a caso:

1. En el caso de la sustitución buscamos demostrar que si $f(y_1, \ldots, y_k)$ y $g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)$ son funciones PL-computables definidas en $\mathcal{T}_{\mathcal{N}}^k$ y $\mathcal{T}_{\mathcal{N}}^n$ respectivamente, entonces la función $h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$ también lo es.

   Al ser todas nuestras funciones PL-computables, tendremos que existirán PL-programas $P_f, P_{g_1}, \ldots, P_{g_k}$ que las computen respectivamente. Tomamos ahora el siguiente programa $P = P_f \cup P_{g_1} \cup \cdots \cup P_{g_k} \cup \{R\}$ siendo $R$ una cláusula que definiremos ahora.

   A la hora de hacer la unión de estos programas, es posible que la cabeza de cláusulas de distintos $P_i$ coincida. No queremos que esto ocurra (podría alterar el comportamiento de nuestras funciones), así que para evitarlo renombraremos las cláusulas que coincidan de modo que entre dos programas distintos no existan predicados iguales. En el Anexo C discutimos esto en profundidad.

   De este modo, siguiendo la nomenclatura del anexo, nos estaríamos refiriendo a $P = P_f \hat{\cup} P_{g_1} \hat{\cup} \ldots \hat{\cup} P_{g_k} \cup \{R\}$. Estamos resaltando este matiz porque se trata de la primera vez que nos hemos encontrado con este problema. A lo largo del trabajo nos lo encontraremos constantemente, por lo que usualmente realizaremos un abuso de notación y nos refiriéndonos a él como la unión.

   Definimos la cláusula $R$ como:

   ```
   R: h(X1,...,Xn,Z) :-
        g1(X1,...,Xn,Y1),
        ...
        gk(X1,...,Xn,Yk),
        f(Y1,...,Yk,Z).
   ```

   Veamos ahora que este programa computa $h$. En la demostración vamos a hacer un pequeño abuso de notación refiriéndonos en ocasiones tanto a la función $f_i$ como al predicado de $P$ que la computa por $f_i$, en vez de utilizar $f_{iP}$ para referirnos al predicado como veníamos haciendo.

   Comenzamos observando que $h(x_1, \ldots, x_n) = z$ si y solo si existen $y_1, \ldots, y_k \in \mathcal{T}_{\mathcal{N}}$ tales que $f(y_1, \ldots, y_k) = z$, $g_1(x_1, \ldots, x_n) = y_1, \ldots, g_k(x_1, \ldots, x_n) = y_k$. Ahora bien, como $P$ computa las funciones $f$ y $g_i$, esto sucederá si y solo si $f(y_1, ..., y_k, z)$, $g_1(x_1, ..., x_n, y_1), \ldots, g_k(x_1, \ldots, x_n, y_k) \in M(P)$ para algunos $y_1, \ldots, y_k \in \mathcal{T}_{\mathcal{N}}$.

   Por otro lado, como $R$ es la única cláusula de $P$ de cuya cabeza $h(x_1, \ldots, x_n, z)$ puede ser instancia, tenemos que $h(x_1, \ldots, x_n, z) \in M(P)$ si y solo si $f(y_1, ..., y_k, z)$, $g_1(x_1, ..., x_n, y_1), \ldots, g_k(x_1, \ldots, x_n, y_k) \in M(P)$ para algunos $y_1, \ldots, y_k \in \mathcal{T}_{\mathcal{N}}$.

   Juntando ambos resultados, hemos llegado a que $h(x_1, \ldots, x_n) = z$ si y solo si $h_P(x_1, \ldots, x_n, z) \in M(P)$, con lo que $P$ computa $h$.

   Una consecuencia directa de este resultado es que si en las condiciones anteriores $f, g_1, \ldots g_k$ son Turing-computables y $f', g'_1, \ldots g'_k$ funciones PL-computables similares a ellas, entonces existe una función $h'$ PL-computable tal que $h'$ es similar a $h$ siendo $h$ la sustitución de las $g_i$ en $f$.

2. Analizamos ahora el caso de la recursiva primitiva. Vamos a probar que si $g(x_1, \ldots, x_n)$ y $h(x_1, \ldots, x_n, y, z)$ (definidas sobre $\mathcal{T}_{\mathcal{N}}^n$ y $\mathcal{T}_{\mathcal{N}}^{n+2}$ respectivamente)son PL-computables, entonces la función $f$ definida por las ecuaciones:

   $f(x_1, \ldots, x_n, cero) = g(x_1, \ldots, x_n)$

   $f(x_1, \ldots, x_n, suc(y)) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$

   también es PL-computable, quedando definida sobre $\mathcal{T}_{\mathcal{N}}^{n+1}$.

Procedemos de un modo similar al apartado anterior. Por ser $g$ y $h$ PL-computables existirán PL-programas que las computen. Sean respectivamente $P_g$ y $P_h$ tales programas, vamos a tomar ahora el PL-programa $P = P_g \cup P_h \cup \{R_1, R_2\}$ siendo $R_1$ y $R_2$ las cláusulas siguientes:

```
R1: f(X1,...,Xncero,Z) :- g(X1,...,Xn,Z).
R2: f(X1,...,XnSuc(cero),Z) :-
      f(X1,...,Xn,Y,V),
      h(X1,...,Xn,Y,V,Z).
```

A la hora de hacer esta unión evitamos como en el apartado anterior la coincidencia de nombres en los predicados.

Veamos que $P$ nos permite computar $f$. Para demostrarlo, aplicaremos inducción matemática sobre el número de veces $n$ que hemos aplicado la constructora *suc* en el último argumento de $f$.

*a*) Para $n = 0$, por definición $f(x_1, \ldots, x_n, cero) = z \Leftrightarrow g(x_1, \ldots, x_n) = z$. Ahora bien, como $P$ computa $g$ está claro que esto sucederá si y solo si $g_P(x_1, \ldots, x_n, z) \in M(P)$.

Por otro lado, como $R_1$ es la única cláusula de $P$ de cuya cabeza $f_P(x_1, \ldots, x_n, z')$ puede ser instancia, tenemos que $f_P(x_1, \ldots, x_n, cero, z') \in M(P)$ si y solo si $g_P(x_1, \ldots, x_n, z) \in M(P)$.

Juntando ahora estos dos últimos resultados, tenemos que $f(x_1, \ldots, x_n, cero) = z \Leftrightarrow f_P(x_1, \ldots, x_n, cero, z) \in M(P)$.

*b*) Pasamos ahora al paso inductivo, en el que probaremos que se verifica para $n+1$ suponiendo que se verifica para $n$.
Tenemos que $f(x_1, \ldots, x_n, suc^{n+1}(cero)) = z \Leftrightarrow h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, suc^n(cero))) = z$. Ahora bien, como $P$ computa la función $h$ tenemos que esto sucederá si y solo si $h_P(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, suc^n(cero)), z) \in M(P)$.

Por otro lado, por nuestra hipótesis de inducción tenemos que $f(x_1, \ldots, x_n, suc^n(cero)) = v$ si y solo si $f_P(x_1, \ldots, x_n, suc^n(cero), v) \in M(P)$.

Por último, vemos que al ser $R_2$ la única cláusula de $P$ de cuya cabeza $f_P(x_1, \ldots, x_n, suc^{n+1}(cero), z)$ puede ser instancia, llegamos a que $f_P(x_1, \ldots, x_n, suc^{n+1}(cero), z) \in M(P)$ si y solo si existe $v$ tal que $f_P(x_1, \ldots, x_n, suc^n(cero), v) \in M(P)$ y $h_P(x_1, \ldots, x_n, y, v, z) \in M(P)$.

Aplicando sobre esta última implicación lo que hemos probado antes, tenemos que $f_P(x_1, \ldots, x_n, suc^{n+1}(cero), z) \in M(P)$ si y solo si $f(x_1, \ldots, x_n, suc(n)) = v$ y $h(x_1, \ldots, x_n, v) = z$. De este manera podemos concluir que $f_P(x_1, \ldots, x_n, suc^{n+1}(cero), z) \in M(P) \Leftrightarrow f(x_1, \ldots, x_n, suc^{n+1}(cero)) = z$.

Dado que $\mathcal{T}_\mathcal{N} = \{suc^n(cero) : n \geq 0\}$ con esto hemos demostrado que $P$ computa $f$, por lo que esta es PL-computable.

Al igual que en el caso anterior, una consecuencia inmediata es que si en las condiciones anteriores $g$ y $h$ son Turing-computables, $g'$ y $h'$ funciones similares PL-computables, entonces existe una $f'$ PL-computable similar a $f$ la recursión primitiva formada a partir de $g$ y $h$.

3. Pasamos por último al caso de las minimizaciones.

Queremos demostrar que si $f(x_1, \ldots, x_n, y)$ es una función PL-computable definida en $\mathcal{T}_{\mathcal{N}}^{n+1}$ entonces la función $g(x_1, \ldots, x_n) = \mu y(f(x_1, \ldots, x_n, y) = 0)$ lo es también.

Recordamos brevemente como funciona el operando $\mu$:

$\mu y(f(x_1, \ldots, x_n, y) = 0)$ = el menor $y$ tal que:

*a*) $f(x_1, \ldots, x_n, z)$ está definido y es distinto de 0 $\forall z \leq y$.

*b*) $f(x, y) = 0$.

si tal $y$ existe. En caso contrario, $\mu y(f(x_1, \ldots, x_n, y) = 0)$ está indefinido.

Pasamos a demostrar que $g$ es PL-computable.

Como $f(x_1, \ldots, x_n, y)$ lo es, entonces existe un PL-programa $P_f$ que la computa. Construimos entonces el PL-programa $P = P_f \cup \{R_1, R_2, R_3\}$ siendo $R_1, R_2$ y $R_3$ las siguientes cláusulas:

```
R1: defNoCero(X1,...,Xn, cero).
R2: defNoCero(X1,...,Xn,suc(Z)):-
      defNoCero(X1,...,Xn,Z),
      f(X1,...,XnZ,suc(V)).
R3: g(X1,...,XnZ) :-
      defNoCero(X1,...,Xn,Z),
      f(X1,...,XnZ,cero).
```

Veamos que $P$ computa $g$.

En primer lugar, vamos a probar que $defNoCero(x_1, ..., x_n, c) \in M(P)$ si y solo si $f(x_1, \ldots, x_n, z)$ está definido y es distinto de *cero* para todo $z \in \mathcal{T}_{\mathcal{N}}$ tal que $cero \leq z < c$ (estableciendo en $z \in \mathcal{T}_{\mathcal{N}}$ el orden $suc^i(cero) < suc^j(cero) \Leftrightarrow i < j$). Lo demostramos al igual que en el apartado anterior por inducción el número de veces que aplicamos la constructora *suc* en $c$.

*a*) Para *cero* resulta trivial. Por un lado, $defNoCero(x_1, ..., x_n, cero) \in M(P)$ para cualquier valor de $x_1, \ldots, x_n$ (siempre es una instancia básica de $R_1$). Por otro, está claro que $f(x_1, \ldots, x_n, z)$ está definido y es distinto de *cero* para todo $z \in \mathcal{T}_{\mathcal{N}}$ tal que $z < cero$, ya que no existe ningún caso que pueda contradecirlo.

*b*) Estudiamos el caso de $suc^{n+1}(cero)$, asumiendo que se verifica para $suc^n(cero)$.

Como $R_2$ es la única cláusula de $P$ de cuya cabeza puede ser instancia $defNoCero(x_1, ..., x_n, suc^n(cero))$, tenemos que $defNoCero(x_1, ..., x_n, suc^{n+1}(cero)) \in M(P)$ si y solo si para algún $z$ $defNoCero(x_1, ...x_n, suc^n(cero))$, $f_P(x_1, ..., x_n, suc^n(cero), suc(z)) \in M(P)$.

Observamos ahora que por computar $P$ la función $f$ tenemos que $f_P(x_1, ..., x_n, suc^n(cero), suc(z)) \in M(P)$ si y solo si $f(x_1, \ldots, x_n, suc^n(cero)) = suc(z)$. Por otro lado, por nuestra hipótesis de inducción tenemos que $defNoCero(x_1, ..., x_n, suc^n(cero)) \in M(P)$ si y solo si $f(x_1, \ldots, x_n, i)$ está definida y es distinta de cero para todo $i$ menor que $suc^n(cero)$.

Combinando ambos resultados hemos llegado a que $defNoCero(x_1, ..., x_n, suc^n(cero))$, $f_P(x_1, ..., x_n, suc^n(cero), suc(z)) \in M(P)$ para algún $z$ si y solo si $f(x_1, \ldots, x_n, i)$ esté definida y sea distinta de *cero* para todo $i$ menor que $suc^{n+1}(cero)$.

Ahora que tenemos este resultado es muy fácil comprobar que $P$ efectivamente computa $g$.

Por la definición de minimización se cumple que $g(x_1, ..., x_n) = z$ si y solo si $f(x_1, \ldots, x_n, z) = 0$ y $f(x_1, \ldots, x_n, v)$ está definido y es distinto de *cero* para todo $v$ tal que $cero \leq v < z$.

Utilizando ahora que $P$ computa $f$ y lo que acabamos de probar acerca de $defNoCero$, tenemos que, en tal caso, $defNoCero(x_1, ..., x_n, z), f_P(x_1, ..., x_n, z, cero) \in M(P)$.

Por otro lado, $R_3$ es la única cláusula de $P$ de la que $g_P(x_1, \ldots, x_n, z')$ puede ser instancia. De este modo, llegamos a que $g_P(x_1, ..., X_n, z) \in M(P)$ si y solo si $defNoCero(x_1, ..., x_n, z')$, $f_P(x_1, ..., x_n, z', cero) \in M(P)$.

Combinando ahora los dos resultados que hemos alcanzado tenemos que $g(x_1, ..., x_n) = z \Leftrightarrow g_P(x_1, \ldots, x_n, z') \in M(P)$.

Finalmente, como $P$ computa la función $g$, queda demostrado que esta función es PL-computable.

Como antes hemos señalado, las funciones recursivas parciales pueden ser básicas ($zero, sig$ y $proy_i$) o haberse formado por sustitución, recursión o minimización de otras funciones recursivas parciales.

Nosotros hemos demostrado que las tres funciones básicas tienen una función PL-computable similar. Luego, hemos probado como la sustitución, recursión y minimalización de funciones PL-computables seguirá siendo PL-computable. Con esto podemos concluir que todas las funciones recursivas parciales tienen una función PL-computable similar.

□

Este es el resultado fundamental que queríamos demostrar en esta sección. Queremos resaltar la gran importancia del mismo, ya que nos prueba la Turing-completitud de las funciones PL-computables. Como hemos visto, para cada función en $\mathcal{T}$ podemos construir una en $\mathcal{L}$ que realiza el mismo cómputo.

De esta manera, tiene sentido estudiar lo que ocurre con las funciones PL-computables y que problemas son capaces de resolver, porque tienen al menos la misma potencia que las máquinas de Turing o las funciones recursivas parciales.

Esta es una de las principales motivaciones de cara a desarrollar nuestra teoría de la PL-recursividad: saber que estamos trabajando con un modelo Turing-completo.

Acabamos la sección añadiendo un sencillo corolario de este resultado.

**Corolario 3.2.1.** *Podemos construir una función total inyectiva $\psi : \mathcal{T} \longrightarrow \mathcal{L}$.*

*Demostración.* Tomamos como $\psi$ la función que a cada $f \in \mathcal{T}$ le asocia una función similar en $\mathcal{L}$. Esta función es total, hemos demostrado en el apartado anterior que siempre va a existir una función similar en $\mathcal{L}$.

Ver que $\psi$ es inyectiva es muy sencillo, supongamos que $\psi(f_1) = \psi(f_2)$. Entonces por ser similares $\psi(f_1)$ y $\psi(f_2)$ a $f_1$ y $f_2$ respectivamente, tenemos que:

$f_1(x_1, ..., x_n) = y \Leftrightarrow \psi(f_1)(\phi(x_1), ..., \phi(x_n)) = \phi(y) \Leftrightarrow f_2(x_1, ..., x_n) = y$

Con lo que $f_1 = f_2$, por lo que $\psi$ es inyectiva.

□

## 3.3. Turing-equivalencia de las funciones PL-computables

En esta sección vamos a incluir una discusión acerca de la Turing-equivalencia de las funciones PL-computables.

Ya hemos demostrado que las funciones PL-computables constituyen un modelo Turing-completo, nos quedaría por demostrar que toda función PL-computable también es Turing-computable para probar que es Turing-equivalente.

En este trabajo no vamos a probar este resultado porque no interviene en nuestro desarrollo de la Teoría de la Computabilidad. Ya hemos demostrado que los PL-programas son capaces de computar tantas funciones como las máquinas de Turing, que es el resultado fundamental de cara a que todo nuestro desarrollo posterior tenga sentido.

El problema al que nos enfrentamos ahora es demostrar que toda función PL-computable es computable por una máquina de Turing (o cualquier otro modelo Turing-equivalente). Una primera dificultad de cara a probar este resultado está en que no hemos desarrollado un modelo de cómputo para nuestros PL-programas. Hemos definido su semántica de una manera completamente abstracta, sin definir ninguna clase de máquina o intérprete que nos permita ejecutar nuestros PL-programas.

De todos modos, una primera idea de cara a construirlo sería hacerlo a través de Prolog, ya que se trata del lenguaje lógico más extendido. Podríamos pensar que un predicado pertenece a la semántica de uno de nuestros PL-programas si al ejecutarlo en Prolog se acepta dicho predicado. Sin embargo, no siempre va a ser así, ya que Prolog utiliza una búsqueda en profundidad determinada por el orden de las cláusulas del programa (que no tenía ninguna importancia en nuestros PL-programas).

Por ejemplo, el PL-programa:

```
cumple(X) :- cumple(X).
cumple(cero).
```

tendrá como semántica $M(P) = \{cumple(cero)\}$. En cambio, al ejecutarlo en un intérprete de Prolog no aceptará el predicado $cumple(cero)$, ya que entraría en una rama infinita al evaluar la primera cláusula.

De este modo, está claro que en un principio Prolog no va a permitirnos construir la semántica de nuestros PL-programas. Sin embargo, es conocido que modificando el tipo de búsqueda que realiza y obligándole a proceder por anchura sí va a ser capaz de computarla.

De manera intuitiva, vemos que la dificultad que ha tenido Prolog de cara a determinar si $cumple(cero)$ formaba parte de la semántica o no estaba en que siempre aplicaba la primera cláusula, a diferencia de nuestros PL-programas, que al estar definidos de forma abstracta tratan de igual manera todas sus cláusulas (no se las evalúa una a una en ningún orden, sino que se consideran las conclusiones inmediatas de todas ellas).

Podemos solucionar este problema adaptando Prolog de manera que no efectúe búsqueda en profundidad, sino en anchura. De esta manera, aseguramos que evalúe si nuestro predicado es consecuencia de todas las cláusulas que se pueden aplicar en él. Esto asegura que vaya a encontrar, en caso de existir, la rama dentro del árbol de búsqueda que prueba el resultado, sin riesgo de quedarse atrapado en una rama infinita situada más a la izquierda.

Aunque no lo demostraremos formalmente, esta variante de Prolog que efectúa su búsqueda en anchura aceptaría únicamente los elementos de la semántica de nuestro PL-programa. Se trataría de este modo de un primer modelo de cómputo para nuestros PL-programas.

Una vez que hemos realizado esta discusión, al haber establecido un modelo de cómputo para

nuestros PL-programas, estaríamos en las condiciones de la tesis de Church-Turing. Las funciones PL-computables podrían calcularse mediante un método efectivo (la variante de Prolog en anchura), por lo que asumiendo la tesis de Turing-Church serían equivalenes a las funciones Turing-computables.

# Capítulo 4

# PL-Programa Universal

Dedicaremos este capítulo a desarrollar un PL-programa universal, es decir, un PL-programa que funcione como intérprete de otros PL-programas. Así, nuestro PL-programa universal, al que denotaremos por $U$, poseerá un predicado distinguido $resuelve\_uno/2$ para el que $resuelve\_uno(P, x) \in M(U) \Leftrightarrow x \in M(P)$ para términos $P, x$ cualesquiera tales que $P$ represente un PL-programa (al que por abuso de notación llamaremos también $P$). El contenido de esta afirmación, así como por supuesto su demostración, se desarrollarán en el resto del capítulo.

Antes de comenzar este desarrollo, nos gustaría resaltar la gran importancia que tiene la existencia de un programa universal o noción equivalente en las distintos modelos de computabilidad (máquinas de Turing, funciones recursivas parciales, máquinas de registros, ...). Este resultado es básico en dichos modelos, ya que se utiliza para demostrar que existen conjuntos no recursivos, que es uno de los principales resultados de la Teoría de la Computabilidad. De este modo, la existencia de un programa universal no va a ser interesante solo por sí misma, sino porque nos proporciona una herramienta básica de cara a demostrar que ciertas funciones no son computables.

En el caso de las funciones PL-computables va a ocurrir lo mismo, como comprobaremos en el próximo capítulo, por lo que resulta fundamental desarrollar un PL-programa universal.

La principal ventaja que va a presentar nuestro PL-programa universal frente a otros metaintérpretes es que los PL-programas van a poder trabajar directamente con otros PL-programas como términos. Esto se debe a la estructura de los PL-programas que desarrollamos en 2.1 y que volveremos a estudiar ahora. Esta estructura nos va a permitir escribirlos fácilmente como términos.

Veremos tanto en este capítulo como en los siguientes cómo esta propiedad nos va a ser de gran ayuda, permitiéndonos construir PL-programas que trabajen con otros PL-programas de forma sencilla.

Este es el motivo por el que, antes de pasar a construir nuestro PL-programa $U$, vamos a desarrollar una sintaxis que nos permita expresar PL-programas como términos. Aunque en un principio esta sintaxis pueda parecer complicada, comprobaremos que nos permite expresar todos los posibles PL-programas. Discutiremos también la razón por la que, por desgracia, la representación es algo más compleja de lo que a primera vista sería posible.

## 4.1. Sintaxis de los PL-programas como términos

Como vimos en nuestro primer capítulo, los PL-programas están muy estructurados. Están constituidos por una lista de cláusulas, formada cada una de ellas por una cabeza (que es un objetivo atómico) y un cuerpo, que a su vez va a ser una lista de objetivos. Del mismo modo, los términos están conformados por su nombre y sus argumentos, que serán también términos.

Esta estructura tan fijada es la que nos va a permitir expresarlos fácilmente como términos de otros PL-programas. No nos va a resultar complicado, solamente vamos a tener que "traducir" los elementos

que conforman el programa (lista de cláusulas, cabeza y cuerpo de cada cláusula,...) a distintas constructoras.

Vamos a comenzar definiendo las constructoras y constantes que vamos a utilizar, para luego pasar a ver cómo van a aplicarse al expresar nuestros PL-programas y desarrollar algún ejemplo de PL-programa traducido.

Tomaremos la siguiente notación.

1. Representaremos listas no vacías usando el constructor $lista(t_1, t_2)$. El primer término se trata de la cabeza de nuestra lista (su primer elemento), mientras que el segundo representará el resto de la lista. Para representar listas vacías utilizaremos la constante *vacia*.

   Aprovechamos el hecho de que toda lista se puede ver como un primer elemento y el resto de la lista, que a su vez será otra lista. De este modo, la lista $[uno, dos]$ se expresaría como $lista(uno, lista(dos, vacia))$.

   Podríamos usar la sintaxis estándar de Prolog para listas, pero hemos preferido no introducir elementos sintácticos nuevos.

2. Representaremos cada una de las cláusulas de nuestro programa mediante el constructor $regla(t_1, t_2)$. El primer término se corresponde con la cabeza de la cláusula mientras que el segundo se trata del cuerpo de la misma.

   El primer término se referirá, por tanto, a un objetivo y el segundo a una lista de objetivos, pudiendo ser esta vacía.

   Nos gustaría resaltar que los objetivos de nuestros PL-programas van a ser siempre términos, por lo que no es necesario disponer de una constructora específica para los objetivos, sino que vamos a poder expresarlos como términos.

3. Definimos los términos de nuestros PL-programas como variables o aplicaciones de constructoras a una lista (que puede ser vacía) de términos. Veamos cómo vamos a expresar cada uno de estos casos.

   Representaremos los términos formados a partir de una constructora aplicada sobre una lista de términos mediante el constructor $const(nombre, lista)$. Su primer argumento se referirá al nombre de la constructora y el segundo a la lista de términos sobre la que se aplica. En caso de ser esta lista vacía se tratará de una constante.

   Por otro lado, vamos a dar un tratamiento especial a las variables.

   En una primera impresión, lo más natural sería representar las variables de los programas lógicos por ellas mismas.

   Esto es lo habitual al construir metaintérpretes de programas lógicos, y constituye un motivo poderoso para su simplicidad, porque facilita enormemente programar operaciones complejas como la unificación. De hecho en muchos metaintérpretes de Prolog queda simplemente realizada por la unificación de Prolog y no es preciso por tanto programarla en absoluto. Podemos encontrar ejemplos de este tipo de metaintérpretes como el intérprete Vanilla, que aparece desarrollado en (Leon S. Sterling, 1994).

   Esa era también nuestra primera impresión al comenzar este trabajo. Por desgracia, las cosas no van a ser tan simples en nuestro caso. Los metaintérpretes que usan variables por variables utilizan siempre algún recurso que queda fuera de la programación lógica que consideramos aquí (como el objetivo *clause* en el caso del intérprete Vanilla). Por tanto, no sirven para nuestro propósito de construir un programa lógico universal, que debe ser un programa lógico puro.

   Incluimos ahora un ejemplo que nos ayude a entender las dificultades. Consideramos el programa lógico con una única cláusula $p(X)$ y supongamos que lo representa mediante la lista de cláusulas $lista(p(X), vacia)$ (nuestra representación es algo más compleja, pero eso es irrelevante aquí).

Nuestro PL-programa universal, que va a definir un predicado $resuelve/2$, debería hacer cierto el objetivo $resuelve(lista(p(X), vacia), p(a))$. Pero, de acuerdo con la semántica de programas lógicos puros, si un objetivo que contiene una variable (como $lista(p(X))$) se satisface, también lo debe hacer cualquier caso particular suyo, como por ejemplo $resuelve(lista(p(b), vacio), p(a))$.

Pero esto está mal, ya que del programa lógico con una única cláusula $p(b)$ no se deduce el objetivo $p(a)$.

Esta discusión explica por qué hemos tenido que optar por una representación básica (ground) de las variables. Una consecuencia de ello es que el PL-programa universal que vamos a desarrollar es más complejo de lo que esperaría un programador Prolog.

A pesar de ello, sigue siendo un programa enormemente más simple que el programa universal de otros modelos de computabilidad como son las máquinas de Turing, máquinas de registros o While programas.

De este modo, una vez que hemos dejado clara nuestra motivación, pasamos a ver la manera en que vamos a traducir nuestras variables como términos ground. Una primera idea sería representarlas por su nombre, pero como veremos al desarrollar nuestro PL-programas universal vamos a tener que estudiar si una variable es igual a otra, de cara a comprobar si dos términos se pueden unificar.

Por este motivo, en vez de representarlas por su nombre, aprovecharemos que tan solo un número finito de variables van a aparecer en cada cláusula del programa. Las podremos numerar dentro de cada regla y referirnos a ellas por su posición en dicha numeración, que denotaremos a través de las constructoras $suc/1$ y $cero/0$. De este modo, si tenemos las variables $X$ e $Y$ en una misma cláusula nos referiremos a ellas por $var(cero)$ y $var(suc(cero))$. Queda claro que $var(cero)$ solo se referirá a $X$ dentro de esta cláusula, si la variable $X$ vuelve a aparecer en alguna otra cláusula nos referiremos a ella según la numeración propia de dicha cláusula.

Con este análisis hemos querido dejar claros los motivos que nos llevan a utilizar las constructoras y constantes que hemos enumerado. Pasamos ahora a una breve discusión acerca de la estructura de los PL-programas que nos permite comprender por qué siempre vamos a poder representarlos a través de las constructoras descritas. Lo hacemos de manera constructiva.

1. Un programa está formado por una lista de cláusulas. De este modo, un programa $P$ queda descrito como $P = [R_1, ..., R_n]$ siendo cada $R_i$ una de sus cláusulas.

   En caso de poder expresar cada cláusula $R_i$ en nuestra nueva sintaxis como $R'_i$, entonces podremos representar $P$ como $lista(R'_1, lista(R'_2, lista(\dots lista(R'_n, vacio)\dots)))$.

2. Cada una de nuestras cláusulas $R_i$ son de la forma $R_i$ : $cabeza$ : $-cuerpo$ siendo $cabeza$ un término y $cuerpo$ una de lista de términos. Si somos capaces de representar ambos, entonces podremos escribir la lista como $regla(cabeza, cuerpo)$.

   Además, al igual que procedimos anteriormente, está claro que si somos capaces de representar un término seremos capaces de representar una lista de ellos. De este modo, para poder demostrar que podemos representar una cláusula (que a su vez implica que podemos representar un programa) tan solo nos queda comprobar que podemos representar un término.

3. Nos queda por último estudiar el caso de los términos. Como ya hemos mencionado pueden tratarse de variables (que ya hemos visto cómo tratar) o construcciones a partir de otros términos.

   Para las construcciones, estas van a ser de la forma $nombre(ter_1, ..., ter_n)$, con lo que vamos a ser capaces de expresarlas mediante la constructora $const$ como $const(nombre, lista(ter'_1, lista(ter'_2, ...)))$. Por inducción estructural, queda claro que todo término va a poder escribirse de este modo.

Acabamos de demostrar que siempre vamos a ser capaces de expresar nuestros PL-programas como términos. De cara a simplificar la notación, vamos a denotar por $\hat{P}$ al término que representa un PL-programa $P$. Del mismo modo, dado un término $t$ nos referiremos a su representación como $\hat{t}$.

Veamos un ejemplo muy sencillo de esta nueva sintaxis, el PL-programa

```
cierto(suc(zero),X).
```

quedaría representado por el término:

$$lista(regla(const(cierto, lista(const(suc, lista(const(zero, vacia), vacia)), lista(var(cero), vacia))), vacia), vacia). \tag{4.1}$$

De este modo, hemos desarrollado en esta sección una sintaxis que nos permite expresar PL-programas como términos. Gracias a ella nuestro PL-programa universal $U$ va a poder trabajar directamente sobre otros PL-programas. Esta sintaxis va a ser fundamental en lo que resta de trabajo, ya que la emplearemos de cara a que nuestros PL-programas trabajen con otros PL-programas.

Desarrollaremos en la siguiente sección nuestro PL-programa universal, que después demostraremos que efectivamente va a comportarse como un metaintérprete de PL-programas.

## 4.2. Desarrollo del PL-programa Universal

Una vez que hemos desarrollado la sintaxis de los PL-programas como términos, pasamos ahora al que es el gran objetivo de este capítulo: construir nuestro PL-programa universal $U$.

Debido a la complejidad del mismo, hemos decidido ir incluyendo las aclaraciones acerca de él a lo largo de su desarrollo, de manera que al lado de cada cláusula aparezca un breve comentario de su funcionamiento. De todas maneras, estas explicaciones se desarrollarán de manera más profunda a lo largo de este capítulo.

Pasamos por tanto a presentar en primer lugar nuestro PL-programa para después demostrar su corrección.

```
/*Dada una lista de términos ground y un PL-programa comprueba si todos ellos
forman parte de su semántica*/
resuelve(vacia, _).

resuelve(lista(A, B), P) :-
    resuelve_uno(A, P),
    resuelve(B, P).

/*Dado un término ground y un PL-programa comprueba si forma parte de su semántica*/
resuelve_uno(Cabeza, P) :-
    existeRegla(Cabeza, P, Cuerpo),
    resuelve(Cuerpo, P).


/*Dado un objetivo A y un programa P, devuelve en Cuerpo el cuerpo
de una instancia ground de una cláusula del programa cuya cabeza unifique con A*/
existeRegla(A, P, Cuerpo) :-
    miembro(P, R),
    esInstancia(R, regla(A, Cuerpo)).


/*Dada una cláusula R y una cláusula ground(sin variables) comprueba si
se trata de una instancia de la primera regla, es decir, si unifica*/
esInstancia(regla(Objetivo, Cuerpo), regla(Objetivo1, Cuerpo1)) :-
    comparaTermino(Objetivo, Objetivo1, Asig),
    comparaLista(Cuerpo, Cuerpo1, Asig),
    asigValida(Asig).
```

```
/*Compara dos términos para ver si el primero puede ser instancia
del segundo, para ello comprueba que el nombre de la constructora es el mismo y despues
compara su lista de argumentos*/
comparaTermino(const(X, Y), const(X, Y1), Asig) :- comparaLista(Y, Y1, Asig).

/*En el caso de tratarse de una variable, comprueba que está asignada al
término que aparece en el segundo término. Para ello comprueba que la
asignación está en la lista de asignaciones Asig*/
comparaTermino(var(X), Y, Asig) :- miembro(Asig, asigna(X, Y)).


/*Dada una lista de términos los compara (coinciden si lo
hacen uno a uno)*/
comparaLista(lista(X,Resto),lista(Y,Resto1),Asig) :-
    comparaTermino(X,Y,Asig),
    comparaLista(Resto,Resto1,Asig).

comparaLista(vacio,vacio,_).


/*Comprueba si una asignación es válida. Para ello revisa que una variable
no haya aparecido dos veces en la lista (se habría asignado dos veces). Lo
comprueba término a término*/
asigValida(vacia).
asigValida(lista(asigna(X, _), Resto)) :-
    noAsignado(X, Resto),
    asigValida(Resto).


/*Dado un natural y una lista de naturales comprueba que el natural no aparece asignado
en la lista*/
noAsignado(var(_), vacia).
noAsignado(var(X), lista(asigna(var(Y), _), Resto)) :-
    distinto(X, Y),
    noAsignado(var(X), Resto).


/*Dados dos naturales comprueba que son distintos*/
distinto(cero, suc(_)).
distinto(suc(_), cero).
distinto(suc(X), suc(Y)) :- distinto(X, Y).


/*Dada una lista comprueba si un elemento forma parte de ella*/
miembro(lista(X, _), X).
miembro(lista(_, Resto), Y) :- miembro(Resto, Y).
```

Vamos a denotar este programa por $U$, comprobaremos ahora que se comporta como un metaintérprete de PL-programas. El siguiente teorema expresa la propiedad fundamental que debe verificar $U$:

**Teorema 4.2.1.** *Dado un PL-programa $P$, se verifica para cualquier objetivo ground $x$:*

$$x \in M(P) \Leftrightarrow resuelve\_uno(\hat{x}, \hat{P}) \in M(U) \tag{4.2}$$

La demostración de este hecho va a ocupar la siguiente sección, en la que estudiaremos cómo se va a comportar cada una de las cláusulas de $U$. Aunque va a ser larga, veremos que esta demostración no va a ser excesivamente complicada. Este hecho representa una de las principales ventajas de los PL-programas, ya que al tener una semántica definida de un modo tan formal nos permite estudiar su significado de un modo sencillo.

Aunque $U$ es bastante complejo comparado con otros metaintérpretes lógicos, debido a que estos desbordan la programación lógica pura, el programa $U$ es mucho más sencillo que el análogo en otros modelos de computabilidad.

Más aun: en estos modelos la construcción de $U$ es tan compleja que es prácticamente inasumible probar que $U$ efectivamente se comporta como un programa universal, sino que se suele dejar asumida "por construcción".

En nuestro caso, sin embargo, vamos a poder probar que $U$ en efecto se comporta como programa universal. Nos gustaría resaltar la gran importancia de este hecho: al utilizar PL-programas como base de nuestra Teoría de la Computabilidad estamos siendo capaces de demostrar con precisión resultados que en la teoría clásica resultan prácticamente inasumibles. Veremos como la demostración no va a ser complicada, aunque sí precisará de bastantes resultados previos acerca de $U$.

Antes de pasar a ella, nos gustaría señalar que aunque hemos distinguido entre los PL-programas $P$ y los términos $t$ y sus representaciones $\hat{P}$ y $\hat{t}$, a partir de ahora por abuso de notación, no distinguiremos entre $P$, $t$ y $\hat{P}$ y $\hat{t}$.

## 4.3. Demostración de la corrección de $U$

Como hemos dicho, el objetivo de esta sección es demostrar que $U$ efectivamente se comporta como un PL-programa universal. Este es uno de los resultados fundamentales de cara a poder desarrollar posteriormente toda nuestra teoría acerca de la PL-computabilidad, por lo que vamos a estudiar en detalle la demostración de este hecho.

No vamos a necesitar apenas recursos de cara a estas demostraciones más allá de algunos sencillos resultados acerca de los PL-programas y distintas técnicas de inducción, como la inducción matemática, estructural y sobre el árbol de derivación, obtenidas de (Hanne Riis Nielson, 2007).

Para poder probar que $U$ se comporta como un PL-programa universal, tratar de hacerlo de manera directa resulta prácticamente inasumible. Por este motivo, vamos a basarnos en una serie de lemas que podemos probar acerca de las cláusulas de este programa, de cara a reunir suficiente información acerca de la semántica de $U$ que nos permita alcanzar el resultado.

Incluiremos estos lemas así como sus demostraciones en el anexo B porque, aunque son importantes de cara a probar la corrección de $U$, sus demostraciones son en su mayoría muy sencillas y repetitivas. Además, estos lemas tan solo van a resultarnos útiles de cara a probar este resultado, por lo que hemos preferido pasarlos al anexo para que no entorpezcan el desarrollo del trabajo, ya que al ser numerosos los lemas que necesitaremos la demostración de todos ellos es larga.

Incluímos ahora el enunciado del último de los lemas que hemos demostrado en el anexo, que es el utilizaremos en nuestra demostración.

**Lema 4.3.1.** *Dado un término ground obj, una lista lista de términos ground y un PL-programa $P$. se tiene que la cláusula $regla(obj, lista)$ es instancia de alguna cláusula de $P$ si y solamente si $existeRegla(obj, P, lista) \in M(U)$.*

Este lema nos proporciona la información acerca de la semántica de $U$ que necesitamos para probar que $U$ se comporta como un PL-programa universal. Lo haremos mediante un razonamiento muy simi-

lar a los desarrollados en el anexo B. Antes de pasar a la demostración, nos gustaría señalar que aunque se trata de una demostración formal, no por ello va a ser complicada. Esta es una de las principales ventajas que nos reportan los PL-programas: al tener definida su semántica de un modo abstracto nos permiten utilizar recursos matemáticos como la inducción para estudiarla.

Esta es una de las principales ventajas que nos aporta el basar nuestro modelo en programas lógicos respecto a otros modelos como las máquinas de Turing o las máquinas de registros: somos capaces de demostrar que nuestro programa universal realmente se comporta como tal. Dado que la existencia de programa universal es uno de los resultados básicos de la Teoría de la Computabilidad de cualquiera de estos modelos, el ser capaces de probar que $U$ es un PL-programa universal es uno de los resultados de mayor importancia de este trabajo.

**Teorema 4.3.1.** *Dado un PL-programa $P$, se verifica para cualquier término ground $obj$:*

$$obj \in M(P) \Leftrightarrow resuelve\_uno(obj, P) \in M(U) \tag{4.3}$$

*Demostración.* Como hicimos para demostrar los lemas previos vamos a probar el resultado por inducción, en este caso lo haremos por inducción sobre el árbol de derivación de la cláusula:

```
R: resuelve_uno(Cabeza,Pr):-
    existeRegla(Cabeza,Pr,Cuerpo),
    resuelve(Cuerpo,Pr)
```

Al ser la cláusula $R$ la única cláusula de $P$ de cuya cabeza $resuelve\_uno(obj, P)$ puede ser instancia, tenemos que $resuelve\_uno(obj, P) \in M(U) \Leftrightarrow$ existe una lista de objetivos básicos $cuerpo$ tal que $existeRegla(obj, P, cuerpo)$, $resuelve(cuerpo, P) \in M(U)$.

Por nuestra hipótesis de inducción, suponemos que para cada uno de los objetivos $obj_i$ de la lista $cuerpo$ obtenida al aplicar $R$ se verifica $obj_i \in M(P) \Leftrightarrow resuelve\_uno(obj_i, P) \in M(U)$. Pasamos ahora a demostrar el resultado.

En primer lugar, observamos que por el lema 4.3.1 tenemos que $existeRegla(obj, P, cuerpo) \in M(U)$ si y solo si $regla(obj, cuerpo)$ es básica y es instancia de alguna de las cláusulas de $P$.

Por otro lado, vamos a demostrar ahora que $resuelve(cuerpo, P) \in M(U)$ si y solo si para cada $obj_i$ de $cuerpo$ se tiene que $obj_i \in M(P)$. Lo demostramos por inducción sobre la longitud de la lista $cuerpo$.

1. Para $cuerpo = vacia$ tenemos que el objetivo $resuelve(cuerpo, P)$ solo puede ser instancia de la cabeza de la cláusula

   ```
   resuelve(vacia,_).
   ```

   De este modo, llegamos a que $solve(cuerpo, prog) \in M(U)$.

   Por otro lado, resulta trivial que todos los $obj_i$ que forman parte de $cuerpo$ cumplen $obj_i \in M(P)$, ya que el cuerpo de la cláusula es vacío.

2. Estudiamos ahora el caso de $cuerpo$ de logitud $k+1$, asumiendo que se verifica el resultado para listas de longitud $k$.

   Comenzamos observando que como $cuerpo$ es de la forma $lista(obj', x')$, entonces la única cláusula de la que $resuelve(cuerpo, P)$ puede ser instancia de su cabeza es

   ```
   resuelve(lista(A,B),Pr):-
       resuelve_uno(A,Pr),
       resuelve(B,Pr).
   ```

De este modo $resuelve(cuerpo, P) \in M(U) \Leftrightarrow resuelve\_uno(obj', P)$, $resuelve(cuerpo', P) \in M(U)$.

Ahora bien, aplicando la hipótesis de inducción sobre la derivación de la cláusula $R$, tenemos que como $obj'$ forma parte de cuerpo entonces se verificará $resuelve\_uno(obj', P) \in M(U) \Leftrightarrow obj' \in M(P)$.

Por otro lado, por la hipótesis de inducción sobre la longitud de *cuerpo*, tenemos que $resuelve(cuerpo', P) \in M(U)$ si y solo si $obj'_i \in M(P)$ para todo $obj'_i$ de $cuerpo'$.

De este modo, juntando ambos resultados llegamos a que $resuelve(cuerpo, P) \in M(U) \Leftrightarrow$ para cada elemento $obj_i$ de la lista $cuerpo$ $obj_i \in M(P)$ .

Volviendo a la expresión que teníamos al comienzo, tenemos que $resuelve\_uno(obj, P) \in M(U) \Leftrightarrow$ existe una lista de objetivos ground $cuerpo$ tal que $existeRegla(obj, P, cuerpo)$, $solve(cuerpo, P) \in M(U)$.

Esto sucederá si y solo si $regla(obj, cuerpo)$ es instancia básica a una cláusula de $P$ y cada objetivo $obj_i$ de $cuerpo$ verifica $obj_i \in M(P)$, como acabamos de demostrar. Como esta es la condición necesaria y suficiente para que $obj$ forme parte de la semántica de $P$, esto nos lleva a que $resuelve\_uno(obj, P) \in M(U) \Leftrightarrow obj \in M(P)$.

□

Acabamos de demostrar el resultado fundamental de este capítulo: existe un PL-programa universal. Además, hemos ido un paso más allá construyendo este programa y comprobando que efectivamente lo es. Ya podemos centrarnos de nuevo en el objetivo fundamental de este trabajo: el estudio de la PL-computabilidad. La existencia de PL-programa universal es uno de los resultados básicos que utilizaremos para encontrar ejemplos de funciones que no sean PL-computables, que es lo que haremos en los próximos capítulos.

Nos gustaría señalar que nuestro teorema 4.2.1 tan solo nos permite estudiar el comportamiento de la semántica cuando $P$ es un término que representa un PL-programa. Veremos en el próximo capítulo como esto nos obliga a reconocer si un término representa un PL-programa antes de poder hacer uso de este teorema.

# Capítulo 5

# Conjuntos PL-recursivos y PL-recusivamente enumerables

Ahora que tenemos definido el concepto de PL-computabilidad, pasamos a centrarnos en cuestiones como las siguientes: ¿existen funciones que no sean PL-computables? ¿cómo son?.

Estas son algunas de las preguntas fundamentales de la Teoría de la Computabilidad. Uno de los objetivos principales de la Teoría de la Computabilidad ha sido establecer criterios que nos permitieran reconocer si un problema no va a poder resolverse mediante una máquina de Turing (o el procedimiento que se corresponda con el modelo de computabilidad tomado). Queremos resaltar lo importante que va a ser este resultado. Que una función no sea computable no quiere decir que no somos capaces de encontrar una máquina de Turing que la compute, sino que va mucho más allá afirmando que esta máquina no existe.

Trataremos en este capítulo de desarrollar estas ideas desde la perspectiva de los PL-programas, trasladando las ideas de la teoría clásica a esta nueva perspectiva.

De esta manera, comenzaremos demostrando la existencia de funciones no PL-computables. Lo haremos en primer lugar de una forma no constructiva utilizando un argumento de cardinalidad. Aunque este resultado es interesante, no nos va a ser muy útil porque, como veremos, no nos da ninguna información acerca de cómo son estas funciones no PL-computables.

Por este motivo dedicaremos lo que resta de capítulo a tratar de encontrar funciones no computables concretas. Para ello desarrollaremos conceptos que van a ser fundamentales en nuestro estudio de la PL-computabilidad, como los conceptos de conjunto PL-recursivo y conjunto PL-recursivamente enumerable.

Estos conceptos se corresponden con la adaptación a la perspectiva de los PL-programas de los conjuntos recursivos y recursivamente enumerables. Estas nociones son fundamentales en la Teoría de la Computabilidad y van a mantener su gran importancia en nuestra nueva perspectiva. Nos hemos basado en las definiciones de conjunto recursivo y conjunto recursivamente enumerable en (Alexander Shen, 2003), aunque vamos a adaptarlas para adecuarlas a nuestro modelo de computabilidad.

Una vez que hayamos desarrollado su definición y algunos resultados básicos acerca de conjuntos PL-recursivos nos embarcaremos en el objetivo principal de este capítulo: encontrar un primer ejemplo de función no PL-computable.

Obtener este primer ejemplo no va a ser una tarea sencilla, por lo que trataremos de desarrollar su demostración con mucho detalle. Como veremos, que este conjunto (al que denotaremos por $H_f$) no sea PL-recursivo va a ser uno de los resultados básicos de este trabajo.

El conjunto $H_f$ no solo va a tener interés por sí mismo, sino que nos va a permitir encontrar otros

muchos ejemplos de conjuntos no PL-recursivos y desarrollar importantes resultados como el Teorema de Rice.

Comenzaremos el capítulo con una pequeña reflexión acerca de los términos de los PL-programas y los conjuntos que vamos a estar interesados en estudiar.

## 5.1. Algunos resultados acerca de los términos de los PL-programas

Comenzamos esta sección estableciendo algunas nociones acerca de los PL-programas y los términos que manejan que queremos aclarar de cara al desarrollo posterior que realizaremos en este trabajo.

Como hemos visto, los PL-programas forman sus términos a partir de una signatura de constructoras fijada $C$, construyendo a partir de ella su conjunto de términos básicos $\mathcal{T}_C$.

Esto nos puede llevar a considerar una primera dificultad: a diferencia del caso de las funciones computables (que están siempre definidas entre naturales), nuestros PL-programas pueden estarlo en conjuntos muy distintos. Sin embargo, esta aparente dificultad no va a ser tal. Aunque los conjuntos $\mathcal{T}_C$ sobre los que trabajan nuestros PL-programas van a ser muy diferentes, vamos a ver que todos poseen una estructura similar que nos permite trabajar sobre ellos de manera general.

Cuando definimos nuestra signatura de constructoras establecimos la condición de que debía ser finita. De esta manera, se cumplirá que $\mathcal{T}_C$ será de cardinalidad a lo sumo numerable por la manera en que se generan los términos. Nos va a interesar el caso de los PL-programas que trabajan con conjuntos de términos infinitos, ya que los finitos son tan sencillos que no nos van a aportar ningún resultado interesante.

Por este motivo, en este capítulo tomaremos para lo que nos queda de trabajo una signatura finita de constructoras $C = \{c_1, \ldots, c_m\}$ tal que $\mathcal{T}_C$ sea infinito. Una condición necesaria para que esto suceda es que $C$ contenga al menos una constante y una constructora que no sea constante (que no tenga aridad 0).

Vamos a incluir una pequeña restricción adicional sobre nuestro $C$. Veremos que en muchas ocasiones vamos a querer asegurarnos de que podamos expresar otros PL-programas en función de los términos de $\mathcal{T}_C$. Para ello, según la sintaxis que adoptamos en la sección 4.1 vamos a necesitar que constructoras como $lista/2, regla/2, const/2, var/1$ o $vacio/0$ formen parte de $C$.

El haber utilizado más constructoras que las estrictamente necesarias se debe únicamente a una cuestión de legibilidad de nuestro código. Podríamos haber expresado todos nuestros PL-programas a través menos constructoras y así haber puesto menos condiciones sobre $C$, pero preferimos mantener la sintaxis que adoptamos en el capítulo anterior ya que resulta más sencilla de manejar. Por este motivo, supondremos que $C$ contiene todas las constructoras que tomamos en el capítulo anterior para expresar PL-programas como términos.

## 5.2. Existencia de funciones no PL-computables

Sean $C$ la signatura de constantes que hemos desarrollado en la anterior sección y $\mathcal{T}_C$ el conjunto formado por sus términos básicos. Vamos a probar que existen funciones $f : \mathcal{T}_C \longrightarrow \mathcal{T}_C$ que no van a ser PL-computables.

Denotaremos por por $\mathcal{F}_C$ al conjunto de todas las funciones definidas en $\mathcal{T}_C \longrightarrow \mathcal{T}_C$ y por $\mathcal{L}_C$ al formado por las que son PL-computables. El resultado que pretendemos alcanzar es, por tanto, que $\mathcal{L}_C \neq \mathcal{F}_C$.

Aunque en un principio nos puede parecer que con este resultado habríamos demostrado todo lo que queríamos saber acerca de funciones no PL-computables, no va a ser así. Esto se debe a que vamos

a demostrar que estos conjuntos son distintos, pero no vamos a obtener ninguna información sobre en qué funciones se diferencian.

Comenzamos estudiando la cardinalidad del conjunto $\mathcal{L}_C$. Nos va a resultar fácil demostrar que es numerable ya que el conjunto de PL-programas definidos sobre $\mathcal{T}_C$ también lo es. De esta manera, estableciendo una inyección entre las funciones computables y los PL-programas que las computan podemos llegar al resultado.

**Lema 5.2.1.** *El conjunto $\mathcal{L}_C$ es de cardinalidad numerable.*

*Demostración.* Como demostramos en 4.1, podemos expresar todos los PL-programas definidos a partir de la signatura de constructoras $C$ como términos de $\mathcal{T}_C$. Esto se debe a que todo PL-programa puede expresarse como término de nuestras constructoras.

De este modo, el conjunto de PL-programas definidos a partir de la constructora $C$ será numerable. Comprobemos ahora que esto implica que también lo será el conjunto de las funciones PL-computables definidas de $\mathcal{T}_C$ en $\mathcal{T}_C$.

Esto se debe a que por la definición de PL-computable que tomamos en 3.1.1 podemos asociar a cada función PL-computable uno de los programas que la computan. Esa asociación define una función entre $\mathcal{L}_C$ y el conjunto de PL-programas que es obviamente inyectiva.

Como hemos podido establecer una inyección de $\mathcal{L}_C$ sobre un conjunto de cardinalidad numerable tenemos que $\mathcal{L}_C$ también será numerable. □

Vamos ahora al caso de todas las funciones definidas de $\mathcal{T}_C$ en $\mathcal{T}_C$. Vamos a probar que este conjunto no es numerable. Para ello vamos a trabajar siguiendo un argumento diagonal. Esta es una de las técnicas más utilizadas en teoría de conjuntos de cara a demostrar que un conjunto no es numerable. Un ejemplo clásico del uso de esta técnica es el argumento diagonal de Cantor, que demuestra que los números reales no son numerables, que puede encontrarse en (Sipser, 2012).

Antes de pasar a la demotración, vamos a aclarar la notación que seguiremos en ella. Nos referiremos por $i$-ésimo término de $\mathcal{T}_C$ al término de la forma $suc^i(cero)$, es decir, al formado por aplicar $i$ veces la constructora *suc* sobre la constante *cero*. Para agilizar la notación tomaremos la abreviatura $n_i$ para referirnos a él.

**Lema 5.2.2.** *El conjunto de las funciones $\mathcal{F}_C$ no es numerable.*

*Demostración.* Supongamos que fuera numerable, veamos que vamos a llegar a una contradicción usando un argumento diagonal.

El conjunto $\mathcal{F}_C$ al ser numerable podrá describirse como una sucesión, de modo que nuestras funciones serán de la forma $f_0, f_1, \ldots$. Denotamos por $c_{i,j}$ al valor de aplicar la función $i$-ésima sobre el $j$-ésimo elemento de $\mathcal{T}_C$ (es decir, sobre $n_j$).

Tomamos la función $g : \mathcal{T}_C \longrightarrow \mathcal{T}_C$ de manera que $g(n_i) \neq c_{i,i} = f_i(n_i)$. Vemos que $g$ está bien definida para todo $i$ (ya que en cada uno nos basta con tomar para $g(n_i)$ un valor distinto a $c_{i,i}$, como puede ser $suc(c_{i,i})$). Esta claro que $g$ no se va a corresponder con ninguna de las funciones $f_i$ ya que diferirá con cada una de ellas al aplicarse sobre $n_i$.

Llegamos así a una contradicción: hemos encontrado una función $g : \mathcal{T}_C \longrightarrow \mathcal{T}_C$ que no forma parte de nuestra sucesión.

Por lo tanto, por reducción al absurdo tenemos que $\mathcal{F}_C$ no puede ser numerable. □

Ahora que ya conocemos las cardinalidades de $\mathcal{L}_C$ y de $\mathcal{F}_C$ es ya prácticamente trivial concluir que van a existir funciones no PL-computables.

Por un lado, hemos visto que existe una cantidad no numerable de funciones de $\mathcal{T}_C$ en $\mathcal{T}_C$. Por otro, tenemos que tan solo va a existir una cantidad no numerable de funciones PL-computables entre estos conjuntos. Es evidente que van a existir más funciones de las que podemos computar, como formalizamos en el siguiente teorema.

**Teorema 5.2.1** (Existencia de funciones no PL-computables)**.** *Existen funciones $f : \mathcal{T}_C \longrightarrow \mathcal{T}_C$ que no son PL-computables. De hecho, hay una cantidad no numerable de ellas.*

*Demostración.* Como hemos visto, $\mathcal{L}_C$ es numerable, mientras que $\mathcal{F}_C$ no lo es. De este modo, podemos afirmar que existen funciones $f : \mathcal{T}_C \longrightarrow \mathcal{T}_C$ que no formarán parte de las PL-computables, y que además serán una cantidad no numerable de ellas. □

Acabamos de demostrar que van a existir funciones que no vamos a ser capaces de computar con nuestros PL-programas. Este es un resultado interesante, pero no nos ha ofrecido ninguna información de cuáles son estas funciones no PL-computables. En lo que resta de capítulo vamos a intentar resolver este problema, encontrando funciones no PL-computables concretas y desarrollando técnicas para demostrar que no lo son.

Esta sección nos ha servido como una motivación de cara a lo que vamos a hacer ahora: ya sabemos que existen funciones no PL-computables, así que centrémonos en encontrarlas. El hecho de que exista una cardinalidad no numerable de funciones no PL-computables nos puede hacer pensar que no va ser complicado encontrarlas. Sin embargo, no va a ser así, como podremos ver en lo que resta de trabajo.

## 5.3. Definición de PL-Recursividad

Sea $C$ la signatura finita de contructoras que definimos en 5.1, $\mathcal{T}_C$ el conjunto de términos generados a partir de ellas.

Vamos a definir ahora dos conceptos que van a resultar fundamentales en nuestra Teoría de la PL-computabilidad. Se trata de los conjuntos PL-recursivos y PL-recursivamente enumerables.

Nuestra intención al definirlos es que se correspondan con lo que conocemos en la teoría clásica como conjuntos recursivos y recursivamente enumerables, pero en nuestro caso definidos sobre el ámbito de los PL-programas. Trataremos en esta sección de definirlos así como de incluir un sencillo resultado sobre ellos que nos permita hacernos una idea de en qué se diferencian.

Comenzamos estableciendo la definición de estos conceptos. Nos gustaría resaltar el hecho de que están definidos en función de $\mathcal{T}_C$. Esto en un principio nos puede resultar extraño, pero al haber tomado un $C$ tan general no va a representar ningún problema.

**Definición 5.3.1** (Conjunto PL-recursivamente enumerable)**.** *Decimos que un conjunto $D \subseteq \mathcal{T}_C^n$ es PL-recursivamente enumerable si existe un PL-programa $P$ con un predicado $p/n$ tal que $\forall (x_1, \ldots, x_n) \in \mathcal{T}_C^n$ se tiene que $(x_1, \ldots, x_n) \in D \Leftrightarrow p(x_1, \ldots, x_n) \in M(P)$.*

**Definición 5.3.2** (Conjunto PL-recursivo)**.** *Decimos que un conjunto $D \subseteq \mathcal{T}_C^n$ es PL-recursivo si se verifica que tanto él como su complementario $\overline{D}$ (que se corresponde con $\mathcal{T}_C^n \backslash D$) son PL-recursivamente enumerables.*

Hemos tomado la terminología de recursivamente enumerable y recursivo para referirnos a estos conceptos porque es habitual en la Teoría de la Computabilidad. Otras denominaciones sinónimas son parcialmente decidible y decidible.

Queda claro que en un principio para ser PL-recursivo estamos pidiendo más que en el caso PL-recursivamente enumerable. Veamos un ejemplo que verifique ambas definiciones.

**Lema 5.3.1.** *El conjunto* $Nat = \{suc^n(cero) | n \geq 0\}$ *es PL-recursivo.*

*Demostración.* Veamos que tanto él como su complemtentario son PL-recursivamente numerables. Estudiamos cada caso de manera independiente. No entraremos en todos los detalles de la demostración porque se trata únicamente de un primer ejemplo de estos conjuntos que no afecta de ningún modo al resto del trabajo.

1. Comenzamos con el caso de $Nat$. Para demostrar que es PL-recursivamente enumerable nos basta con encontrar un PL-programa $P$ cuyo predicado $enNat$ verifique $\forall x \in \mathcal{T}_C^n$ que $x \in Nat \Leftrightarrow enNat(x) \in M(P)$.

   Tomamos el PL-programa siguiente:

   ```
   enNat(cero).
   enNat(suc(X) :- enNat(X).
   ```

   Por construcción parece evidente que el predicado $enNat$ de este programa va a cumplir la condición. De todos modos, podría probarse formalmente mediante un razonamiento por inducción.

2. Para $\overline{Nat}$ en un principio nos puede parecer que la demostración es más complicada. Sin embargo, observando la forma de nuestro conjunto $\mathcal{T}_C$ vamos a ver como sigue siendo muy sencilla.

   Tenemos que $C$ está formado por las constructoras $cero, suc, c_2, \ldots, c_m$. De este modo, los elementos que no formen parte de $Nat$ serán:

   *a*) Toda constructora distinta a *suc* aplicada sobre otros términos cualquiera.

   *b*) La constructora *suc* aplicada sobre todo término que no pertenezca a $Nat$.

   Ahora que hemos localizado que términos queremos reconocer, construir nuestro PL-programa es muy sencillo. Suponemos que el predicado que reconoce la pertenencia es $enNoNat$ y que cada constructora $c_i$ de $C$ tiene aridad $n_i$.

   ```
   enNoNat(c1(X1,...,Xni)).
   ...
   enNoNat(cm(X1,...,Xnm)).
   enNoNat(suc(X) :- enNoNat(X).
   ```

   Demostrar formalmente que este programa computa el predicado $enNoNat$ es sencillo (al igual que en el caso anterior va a requerir inducción), pero dado que solo se trata de un ejemplo de nuevo no hemos entrado en tanta formalidad.

□

Con este primer ejemplo queríamos mostrar cuál sería el modo de proceder de cara a demostrar que un conjunto es PL-recursivo. Vemos que para probarlo hemos tenido que trabajar tanto sobre el conjunto como sobre su complementario, teniendo que probar dos veces que un conjunto es PL-recursivamente enumerable.

Vamos ahora a tratar de dar una caracterización de la PL-recursividad más directa, que no pase por comprobar si el complementario es PL-recursivamente enumerable. Además esta caracterización nos va a ayudar a entender la diferencia entre estos dos conceptos.

Lo que vamos a hacer en el siguiente lema es desarrollar una definición equivalente de la PL-recursividad de un conjunto, que va a estar basada en la computabilidad de la función característica del

conjunto. De este modo, además de facilitarnos la demostración de que un conjunto es PL-recursivo, esta demostración muestra la relación entre los conjuntos PL-recursivos y las funciones PL-computables.

**Proposición 5.3.1.** *Un conjunto $D \subseteq \mathcal{T}_C^n$ es PL-recursivo si y solo si la función característica de $D$ es PL-computable.*

*Es decir, si existe un PL-programa $P$ que verifica:*

$$\forall (x_1,\ldots,x_n) \in \mathcal{T}_C^n \begin{cases} car(x_1,\ldots,x_n,s(zero)) \in M(P), & sii \quad (x_1,\ldots,x_n) \in D \\ car(x_1,\ldots,x_n,zero) \in M(P), & sii \quad (x_1,\ldots,x_n) \notin D \end{cases}$$

*Demostración.* Veamos que se cumplen cada una de las dos implicaciones.

1. Comenzamos con la implicación a derechas.

   Tenemos que tanto $D$ como su complementario $\overline{D}$ son PL-recursivamente enumerables. De este modo, existirán PL-programas $P_D$ y $P_{\overline{D}}$ que computen los predicados $enD$ y $enNoD$, de manera que cumplan respectivamente la definición 5.3.1.

   Tomamos $P = P_D \cup P_{\overline{D}} \cup \{R_1, R_2\}$ siendo $R_1$ y $R_2$ las siguientes cláusulas:

   $R_1$: `car(X1,...,Xn,cero) :- enNoD(X1,...,Xn).`

   $R_2$: `car(X1,...,Xn,suc(cero)) :- enD(X1,...,Xn).`

   Veamos que $P$ computa la función característica de $D$. Para simplificar la notación utilizaremos $\overline{x}$ para referirnos los términos $(x_1,\ldots,x_n) \in \mathcal{T}_C^n$.

   Comenzamos observando que dado que $P$ computa $enD$ tenemos que se verifica que $\overline{x} \in D \Leftrightarrow enD(\overline{x}) \in M(P)$. Por otro lado, como $P$ también computa $enNoD$ tenemos también que $\overline{x} \notin \overline{D} \Leftrightarrow enNoD(\overline{x}) \notin M(P)$.

   Como $R_1$ y $R_2$ son las únicas cláusulas de $P$ con $carD$ en su cabeza, entonces se verificará que $enD(\overline{x}) \in M(P) \Leftrightarrow car(\overline{x}, suc(zero)) \in M(P)$ y que $enNoD(\overline{x}) \in M(P) \Leftrightarrow car(\overline{x}, zero) \in M(P)$.

   Uniendo estas expresiones a lo que ya teníamos, llegamos a que $\overline{x} \in D \Leftrightarrow car(\overline{x}, suc(cero)) \in M(P)$ y $\overline{x} \notin D \Leftrightarrow car(\overline{x}, cero) \in M(P)$, con lo que $P$ computa la función característica de $D$.

   Por lo tanto, *car* es PL-computable como queríamos demostrar.

2. Pasamos a la otra implicación, en la que probaremos que $D$ es recursivamente enumerable. La demostración para $\overline{D}$ es análoga y por eso no la incluimos.

   Sea $P$ un programa que computa la función característica de $D$ mediante el predicado *car*, tomamos entonces $P_D = P \cup \{R\}$ siendo $R$ la cláusula siguiente:

   R: `en(X1,...,Xn) :- car(X1,...,Xn,suc(cero)).`

   Comprobemos que $P_D$ computa en el predicado *en* que buscamos. Al igual que en el apartado anterior tomamos $\overline{x}$ para referirnos a los términos $(x_1,\ldots,x_n) \in \mathcal{T}_C^n$.

   Vemos que por computar $P_D$ la función *car* tenemos que dado $\overline{x} \in \mathcal{T}_C^n$ se verifica que $\overline{x} \in D \Leftrightarrow car(\overline{x}, suc(zero)) \in M(P_D)$.

Ahora bien, por definición de modelo mínimo, al ser $R$ la única cláusula con el predicado $en$ en su cabeza, tenemos que $car(\overline{x}, suc(zero)) \in M(P_D)$ si y solo si $en(\overline{x}) \in M(P_D)$.

De esta manera, hemos llegado a que $\overline{x} \in D \Leftrightarrow en(\overline{x}) \in M(P_D)$, con lo que $D$ es PL-recursivamente enumerable.

□

Reflexionemos ahora brevemente sobre estas nuevas definiciones.

Vemos que en un sentido informal, la diferencia entre un conjunto PL-recursivamente enumerable y uno PL-recursivo está en que mientras en el primero solamente podemos obtener información sobre qué elementos están en él (mediante el predicado $en$), en el PL-recursivo podemos saber qué elementos están y cuáles no (a través del programa que computa su función característica).

Esta diferencia en un principio nos puede parecer poco importante, pero vamos a comprobar que no es así encontrando conjuntos PL-recursivamente numerables pero no PL-recursivos.

La razón que hay detrás de esto es que no tenemos modo de expresar la negación en nuestros PL-programas, es decir, no existe el predicado $not$ en ellos.

## 5.4. Propiedades de los conjuntos PL-recursivamente enumerables y PL-recursivos

Dedicaremos esta sección a estudiar algunas propiedades sencillas que van a verificar los conjuntos PL-recursivos y los PL-recursivamente enumerables. Nos centraremos en estudiar si estas familias de conjuntos van a ser cerradas respecto a algunas operaciones como la intersección, unión o complemento.

Vamos a hacerlo caso a caso. Comenzaremos viendo aquellas propiedades que van a ser ciertas. Desarrollaremos en todas ellas la demostración de este hecho y veremos algunas de sus consecuencias.

Para los casos en los que la operación no es cerrada, que se especificarán al final de la sección, todavía no estamos en condiciones de mostrar contraejemplos. Esto se debe a que aún no hemos desarrollado ningún ejemplo de conjunto no PL-recursivo ni no PL-recursivamente enumerable.

De todos modos, indicaremos a modo de observación qué conjuntos nos podrían servir de contraejemplos, aunque no los desarrollemos (lo haremos más adelante).

Comenzamos estudiando la unión e intersección finita de conjuntos PL-recursivamente enumerables. Comprobaremos que se va a tratar de una operación cerrada, construyendo un PL-programa que compute el predicado $en$ que verifica 5.3.1 del conjunto que genera la unión o intersección.

Ambas demostraciones mantienen una estructura muy similar y en ellas es clave el hecho de que estamos uniendo (o intersecando) un número finito de conjuntos.

**Proposición 5.4.1.** *La unión finita de conjuntos PL-recursivamente enumerables es PL-recursivamente enumerable.*

*Demostración.* Sean $A_1, \ldots, A_k \subseteq \mathcal{T}_C^n$ conjuntos PL-recursivamente enumerables. Veamos que su unión $B = A_1 \cup \cdots \cup A_k$ también es PL-recursivamente enumerable.

Por ser nuestros $A_i$ PL-recursivamente enumerables, tenemos que para cada uno de ellos existirá un PL-programa $P_i$ que compute el predicado $enA_i$ correspondiente. Tomamos ahora el programa $P = P_1 \cup \cdots \cup P_k \cup \{R_1, \ldots, R_k\}$ siendo las cláusulas $R_i$ de la forma siguiente:

`R`$_i$`: en(X`$_1$`,...,X`$_n$`) :- enA`$_i$`(X`$_1$`,...,X`$_n$`).`

Queremos resaltar que podemos realizar esta unión porque se trata de un número finito de conjuntos. Esto hace que al realizar la unión de sus programas y las cláusulas $R_i$ el programa $P$ que obtenemos tenga un número finito de cláusulas. Dado que nuestros PL-programas son finitos, esta demostración no sería válida para uniones infinitas.

Ahora veamos que el predicado *en* computado por $P$ nos sirve para demostrar que $B$ es PL-recursivamente enumerable. Al igual hemos hecho antes, tomamos tomamos $\overline{x}$ para referirnos a los términos $(x_1 \ldots, x_n) \in \mathcal{T}_C^n$.

Tenemos que $\overline{x} \in B \Leftrightarrow \overline{x} \in A_i$ para cierto $i$ entre 1 y $k$. Como $P$ computa su correspondiente predicado $enA_i$ tenemos que esto sucederá si y solo si $enA_i(\overline{x}) \in M(P)$.

Ahora bien, por ser $M(P)$ modelo mínimo y las $R_i$ las únicas cláusulas con $enB$ en su cabeza, tenemos que para cierto $i$ $enA_i(\overline{x}) \in M(P)$ si y solo si $enB(\overline{x}) \in M(P)$.

De esta manera hemos llegado a que $\overline{x} \in B$ si y solo si $enB(\overline{x}) \in M(P)$. Podemos así afirmar que $P$ computa el predicado *en* que queríamos y por tanto $B$ es PL-recursivamente enumerable.

□

Una consecuencia directa de este resultado (aunque podríamos haberla demostrado muy fácilmente sin él) es que todo conjunto finito es PL-recursivamente enumerable. Dado que un conjunto finito puede expresarse como la unión finita de conjuntos de un único elemento tan solo nos queda demostrar que cualquier conjunto formado únicamente por un término es PL-recursivamente enumerable.

**Corolario 5.4.1.** *Todo conjunto finito $A$ es PL-recursivamente enumerable.*

*Demostración.* Sea el $A$ conjunto finito, podremos expresarlo como $A = \{\overline{t_1}, \ldots, \overline{t_k}\}$ siendo los $\overline{t_i}$ términos de $\mathcal{T}_C^n$.

Demostraremos ahora que el conjunto de un único elemento $A_i = \{\overline{t}\}$ va a ser siempre PL-recursivamente enumerable, siendo $\overline{t} = (t_1, \ldots, t_n)$. Para ello ya nos basta tomar con el PL-programa $P$ formado por la cláusula:

`R: en(t`$_1$`,...,t`$_n$`).`

Evidentemente este programa verifica que $en(\overline{x}) \in M(P) \Leftrightarrow \overline{x} = \overline{t}$, por lo que nos sirve para probar que $A_i$ es PL-recursivamente enumerable.

Concluimos la demostración observando que $A = A_1 \cup \cdots \cup A_k$ es la unión finita de los $A_i$ con $i = 1, \ldots, k$. Por lo tanto aplicando la proposición que acabamos de desarrollar tenemos que es PL-recursivamente enumerable. □

Resaltemos que la proposición 5.4.1 tan solo es válida para uniones finitas de conjuntos.

En caso de serlo también para uniones numerables podríamos haber seguido un razonamiento similar al de este corolario que nos afirmara que todo conjunto numerable es PL-recursivamente enumerable. Como $\mathcal{T}_C$ es numerable, tendríamos que todo subconjunto suyo sería PL-recursivamente enumerable.

Vamos a comprobar que esto no va a ser así encontrando conjuntos no PL-recursivamente enumerables como $\overline{H_f}$ (definido en 5.5.1) o $IG_f$ (definido en 6.1.2). De este modo, que existan estos conjuntos nos sirve como contraejemplo a que la unión numerable de conjuntos PL-recursivamente enumerables es PL-recursivamente enumerable.

Pasamos ahora al caso de la intersección finita de conjuntos PL-recursivamente numerables. Seguiremos en él un razonamiento similar al caso de la unión finita para demostrar que también va a ser

cerrada.

**Proposición 5.4.2.** *La intersección finita de conjuntos PL-recursivamente enumerables es un conjunto PL-recursivamente enumerable.*

*Demostración.* Sean $A_1, \ldots, A_k \subseteq \mathcal{T}_C^n$ conjuntos PL-recursivamente enumerables, demostremos que su intersección $B = A_1 \cap \cdots \cap A_k$ también lo es.

Para cada $i = 1, \ldots, k$ como $A_i$ es PL-recursivamente enumerable, tenemos que para cada uno de ellos existirá un PL-programa $P_i$ que compute el predicado $enA_i$. Tomamos ahora $P = P_1 \cup \cdots \cup P_k \cup \{R\}$ siendo $R$ la cláusula:

```
R: en(X1,...,Xn) :-
      enA1(X1,...,Xn),
      ...
      enAk(X1,...,Xn).
```

Al igual que en el caso anterior, es importante resaltar que podemos construir este programa porque se trata de un número finito de conjuntos.

Veamos que el predicado $en$ nos vale para demostrar que $B$ es PL-recursivamente numerable.

Como $B$ es la intersección de estos conjuntos se verifica que $\overline{x} \in B \Leftrightarrow \overline{x} \in A_i$ para todo $i$ entre 1 y $k$. Ahora bien, como $P$ computa los predicados $enA_i$ de cada conjunto, tenemos que $\overline{x} \in A_i \Leftrightarrow enA_i(\overline{x}) \in M(P)$.

Juntando ambos resultados llegamos a que $\overline{x} \in B$ si y solo si para todo $i$ tenemos que $enA_i(\overline{x}) \in M(P)$. Como $R$ es la única cláusula de $P$ con el predicado $en$ en su cabeza tenemos finalmente que $en(\overline{x}) \in M(P)$ si y solo si $enA_i(\overline{x}) \in M(P)$ en todo $i$ entre 1 y $k$.

Hemos llegado a que $\overline{x} \in B \Leftrightarrow en(\overline{x}) \in M(P)$, por lo que $P$ nos permite afirmar que $B$ es PL-recursivamente enumerable.

□

Al igual que en el caso de la unión hemos llegado a que la PL-recursividad enumerable se preserva por intersecciones finitas.

Esto no es cierto para intersecciones infinitas. Para demostrarlo podríamos seguir un razonamiento análogo al caso anterior de la unión infinita. Nos bastaría con demostrar que el conjunto $\mathcal{T}_C \setminus \{x\}$ es PL-recursivamente enumerable. Entonces, en caso de que la intersección infinita fuera PL-recursivamente enumerable, tendríamos que todo conjunto lo sería. Como hemos visto, tenemos contraejemplos de este hecho como $\overline{H_f}$ o $IG_f$.

Por último, observemos que la PL-enumerablilidad recursiva tampoco es cerrada con respecto al complemento. De nuevo nos sirve $H_f$ de contraejemplo.

Nos centramos ahora en los conjuntos PL-recursivos, donde vamos a poder probar resultados similares de una manera sencilla para el caso de la unión e intersección. Pero antes de verlos pasamos a un resultado respectivo al complemento de conjuntos PL-recursivos que se desprende inmediatamente de su definición.

**Lema 5.4.1.** *Un conjunto $A$ es PL-recursivo si y solo si su complemento $\overline{A}$ lo es.*

*Demostración.* Es trivial a partir de la definición de conjunto PL-recursivo.

□

Vamos ahora a los casos de la unión e intersección finita de conjuntos PL-recursivos.

En primer lugar demostraremos que la familia de conjuntos PL-recursivos es cerrada respecto a estas operaciones. Para ello usaremos las distintas proposiciones que acabamos de demostrar acerca de estas operaciones y los conjuntos PL-recursivamente enumerables. Nos basaremos para ello en la definición de la PL-recursividad de un conjunto como que él y su complementario sean PL-recursivamente enumerables.

Sin embargo, esta demostración no nos va a proporcionar un PL-programa que compute la función característica del conjunto resultante. Por este motivo dedicaremos el final de esta sección a desarrollarlo, siguiendo un esquema similar al llevado a cabo al generar nuestros predicados *en* para la unión o intersección de conjuntos.

Pero antes de esto, pasamos a la demostración de que la familia de conjuntos PL-recursivos es cerrada respecto a la unión e intersección finita.

**Lema 5.4.2.** *La unión finita de conjuntos PL-recursivos es PL-recursiva.*

*Demostración.* Tenemos $A_1, \ldots, A_k$ conjuntos PL-recursivos, buscamos demostrar que el conjunto $B = A_1 \cup \cdots \cup A_k$ es PL-recursivo. Para ello nos basta con probar que tanto $B$ como $\overline{B}$ son PL-recursivamente enumerables.

Comenzamos viendo que por definición de conjunto PL-recursivo, tenemos que tanto $A_1, \ldots, A_k$ como $\overline{A_1}, \ldots, \overline{A_k}$ serán PL-recursivamente enumerables.

Ahora bien, como $B = A_1 \cup \cdots \cup A_k$ es unión finita de conjuntos PL-recursivamente enumerables, aplicando la proposición 5.4.1 tenemos que $B$ será PL-recursivamente enumerable.

Para el caso del complemento partimos de que $\overline{B} = \overline{A_1 \cup \cdots \cup A_k} = \overline{A_1} \cap \cdots \cap \overline{A_k}$. Dado que es intersección finita de conjuntos PL-recursivamente enumerables, por la proposición 5.4.2 también será PL-recursivamente enumerable.

Como hemos visto que tanto $B$ como $\overline{B}$ son recursivamente enumerables tenemos que $B$ es PL-recursivo.

□

Demostraremos ahora el mismo resultado para la intersección finita de conjuntos PL-recursivos. En este caso la demostración va a ser aun más sencilla al poder basarnos en los dos resultados que acabamos de demostrar acerca de la PL-recursividad del complemento y la unión, tan solo vamos a tener que expresar esta intersección finita en términos de uniones e intersecciones.

**Lema 5.4.3.** *La intersección finita de conjuntos PL-recursivos es PL-recursiva.*

*Demostración.* Sean $A_1, \ldots, A_k$ conjuntos PL-recursivos, veamos que $B = A_1 \cap \cdots \cap A_k$, es PL-recursivo.

Tenemos que $B$ será PL-recursivo $\Leftrightarrow$ $\overline{B}$ lo es. Ahora bien, como $\overline{B} = \overline{A_1}, \cup \ldots, \cup \overline{A_k}$, aplicando los dos lemas anteriores llegamos a que $\overline{B}$ será PL-recursivo al ser unión finita de conjuntos PL-recursivos.

Del hecho de que $\overline{B}$ sea PL-recursivo podemos concluir que $B$ también lo es. □

Ya hemos encontrado todos los resultados que queremos probar en esta sección. Sin embargo, en estas últimas demostraciones no hemos computado la función característica de estos conjuntos. Nos hemos limitado a aprovechar la definición de PL-recursividad de un conjunto como que tanto él como su complementario sean PL-recursivamente enumerables.

Completamos ahora estas demostraciones construyendo PL-programas que computen sus funciones características. Este resultado no es necesario para probar que son PL-recursivos como hemos visto, pero lo incluimos porque nos va a permitir comprender mejor por qué en el caso finito de la unión e

intersección vamos a poder construir las funciones características, mientras que en el infinito no vamos a ser capaces. El único interés de estos lemas son los PL-programas que vamos a construir, no nos van a aportar ningún resultado nuevo acerca de la PL-recursividad de conjuntos. Por lo tanto, no vamos a razonar apenas acerca de ellos, limitándonos a desarrollar los programas que computen sus funciones características y demostrando que lo hacen.

**Ejemplo 5.4.1.** *Construcción de un PL-programa que compute la función característica del complemento de un conjunto PL-recursivo.*

*Demostración.* Sea $A$ conjunto PL-recursivo y $P_A$ el PL-programa que compute su función característica.

Construyamos ahora un PL-programa $P_B$ que compute la función característica de $B = \overline{A}$. Tomamos para ello $P_B = P_A \cup \{R_1, R_2\}$ siendo estas las cláusulas:

```
R1: car(X1,..., Xn, cero) :- carA(X1,..., Xn, suc(cero)).
R2: car(X1,..., Xn, suc(cero)) :- carA(X1,..., Xn ,cero).
```

Veamos que $P_B$ computa *car*. Comprobaremos los dos casos posibles según $\overline{x} = (x_1, \ldots, x_n)$ pertenezca o no a $B$.

1. Tratamos ahora el caso $\overline{x} \in B$. Está claro que por ser $A$ su complemento tenemos que $\overline{x} \in B \Leftrightarrow \overline{x} \notin A$.

   Por computar $P_B$ la función característica de $A$ se verifica que $\overline{x} \in A \Leftrightarrow carA(\overline{x}, suc(zero)) \in M(P_B)$.

   Ahora bien, como $R_2$ es la única cláusula de $P_B$ cuya cabeza encaja con $car(\overline{x}, suc(zero))$, tenemos que $carA(\overline{x}, suc(zero)) \in M(P_B)$ si y solo si $car(\overline{x}, cero) \in M(P_B)$.

   De esta manera, hemos alcanzado que $\overline{x} \in B \Leftrightarrow car(\overline{x}, cero) \in M(P_B)$.

2. Para el caso $\overline{x} \in B$ el razonamiento es análogo utilizando la cláusula $R_1$.

De este modo, concluimos que $P_B$ computa la función característica de $B$.

□

**Ejemplo 5.4.2.** *Construcción de un PL-programa que compute función la característica de la unión finita de conjuntos PL-recursivos.*

*Demostración.* Sean $A_1, \ldots, A_k$ conjuntos PL-recursivos, tomamos $P_1, \ldots, P_k$ los PL-programas que computan sus respectivas funciones características $carA_i$.

Construyamos ahora un PL-programa $P_B$ que compute la función característica de $B = A_1 \cup \cdots \cup A_k$. Tomamos para ello el PL-programa $P_B = P_1 \cup \cdots \cup P_k \cup \{R, R_1, \ldots, R_k\}$ siendo estas cláusulas las siguientes:

```
R: carB(X1,..., Xn,cero) :-
      carA1(X1,..., Xn,cero),
      ...
      carAk(X1,..., Xn,cero),

R1: carB(X1,..., Xn,suc(cero)) :- carA1(X1,..., Xn,suc(cero)).
      ...
Rk: carB(X1,..., Xn,suc(cero)) :- carAk(X1,..., Xn,suc(cero)).
```

Comprobar que *carB* efectivamente computa la función característica de $B$ nos va a resultar sencillo. Para ello analizamos los dos casos posibles según $\overline{x} = (x_1, \ldots, x_n)$ pertenezca o no a $B$:

1. Estudiamos ahora el caso de $\overline{x} \in B$.

   Comenzamos observando que esto sucede si y solo si existe un $i$ tal que $\overline{x} \in A_i$.

   Además, tenemos que por computar $P_B$ la función característica de $A_i$ se verifica que $\overline{x} \in A_i \Leftrightarrow carA_i(\overline{x}, suc(cero)) \in M(P_B)$.

   Ahora bien, como las distintas $R_i$ son las únicas cláusulas de $P_B$ de cuyas cabezas puede ser instancia $carB(\overline{x}, suc(cero))$ tenemos que se verifica $carB(\overline{x}, suc(cero)) \in M(P_B)$ si y solo si existe un $i$ tal que $carA_i(\overline{x}, suc(cero)) \in M(P_B)$.

   Acabamos de demostrar que $\overline{x} \in B \Leftrightarrow carB(\overline{x}, suc(cero)) \in M(P_B)$, como queríamos.

2. Procedemos ahora para el caso $\overline{x} \notin B$.

   Si $\overline{x} \notin B$ si y solo si $\overline{x} \notin A_i$ para cada uno de los $A_i$.

   Como para cada uno de estos conjuntos $P_B$ computa su función característica tendremos, por tanto, que $\overline{x} \notin A_i \Leftrightarrow carA_i(\overline{x}, cero) \in M(P_B)$.

   Observamos ahora que $R$ es la única cláusula de $P_B$ de cuya cabeza puede ser instancia $c\overline{x}, zero)$, por tanto, se verifica que $carB(\overline{x}, cero) \in M(P_B)$ si y solo si $carA_1(\overline{x}, cero), ..., carA_k(\overline{x}, cero) \in M(P_B)$.

   De este modo, hemos llegado a que $\overline{x} \notin B \Leftrightarrow carB(\overline{x}, cero) \in M(P_B)$.

Acabamos de demostrar que $P_B$ computa la función característica de $B$, como queríamos. □

**Ejemplo 5.4.3.** *Construcción de un PL-programa que compute la función caracterítica de la intersección finita de conjuntos PL-recursivos.*

*Demostración.* En este caso, vamos a desarrollar el programa apoyándonos en los dos resultados que acabamos de demostrar.

Tenemos que $A_1 \cap \cdots \cap A_k = \overline{\overline{A_1} \cup \cdots \cup \overline{A_k}}$. Entonces, dado que hemos expresado la intersección a través de complementos y uniones, ya estamos capacitados para desarrollar el PL-programa $P$ que queremos porque conocemos como construirlo en estos casos.

Este PL-programa será el formado a través de $P_B = P_1 \cup \ldots P_k \cup \{R, R_1, \ldots, R_k\}$ siendo los $P_i$ los PL-programas que computan las funciones características de los distintos conjuntos $A_i$ y las cláusulas siguientes:

```
R: carB(X1,..., Xn, suc(cero)) :-
      carA1(X1,..., Xn,suc(cero)),
      ...
      carAk(X1,..., Xn,suc(cero)).

R1: carB(X,cero) :- carA1(X1,..., Xn,cero).
      ...
Rk: carB(X1,..., Xn,cero) :- carAk(X1,..., Xn,cero).
```

La demostración de que este programa computa efectivamente la función característica pedida es muy similar a las anteriores. □

Con estas construcciones ya habríamos terminado de estudiar las principales operaciones respecto de las que van a ser cerradas las familias de conjuntos PL-recursivamente enumerables y PL-recursivos. En el resto de casos que no hemos cubierto no va a ser así, pero aún no estamos en condiciones de encontrar contraejemplos que lo muestren (aún no tenemos siquiera ningún conjunto no PL-recursivo). De todos modos, hemos señalado algunos de los ejemplos como $IG_f$ que más adelante desarrollaremos.

Vamos a centrarnos ahora en encontrar un primer ejemplo de conjunto no PL-recursivo. Esta no va a ser tarea tan sencilla como las que venimos haciendo, sino que va a requerir técnicas de diagonalización para demostrar que efectivamente no es PL-recursivo.

## 5.5. No PL-recursividad de $H_f$

Vamos a dedicar esta sección a desarrollar un primer ejemplo de conjunto no PL-recursivo. Como vamos a ver, se trata de un conjunto formado por PL-programas, con los que estamos aprovechando la capacidad de estos de escribirse como términos según vimos en la sección 4.1.

Trabajamos como hemos venido haciendo sobre la signatura de constructoras $C$ definida anteriormente. Por lo tanto, lo que pretendemos hacer en esta sección es encontrar un subconjunto de $\mathcal{T}_C$ al que denotaremos por $H_f$ que no sea PL-recursivo.

Dado que los conjuntos PL-recursivos son la traducción a la perspectiva de los PL-programas de los conjuntos recursivos, una primera idea que podríamos tener de cara a resolver este problema es centrarnos en alguno de los ejemplos clásicos de conjuntos no recursivos de la Teoría de la Computabilidad. Si conseguimos trasladarlo al ámbito de la PL-programación parece que efectivamente tampoco será PL-recursivo.

Un primer ejemplo en el que podríamos pensar es el conocido como Problema de Parada. Este resultado nos permitía afirmar que el conjunto $H := \{(i, x)|$ el programa $i$ para al ejecutarse con la entrada $x\}$ no es recursivo. Queremos remarcar como en este caso nos estamos refiriendo a programas sobre máquinas de Turing (que se codifican como naturales), no a PL-programas.

De cara a trasladarlo al ámbito de los PL-programas nos encontramos con una primera dificultad. No hemos definido en el ámbito de la PL-programación qué significa que un programa pare, y dada la naturaleza abstracta de nuestra semántica no vamos a querer hacerlo. Por este motivo, en nuestro ejemplo trasladaremos la propiedad de que un programa pare para una entrada $x$ en que $x$ forme parte de la semántica de un PL-programa en uno de sus predicados.

De este modo, el conjunto que estaríamos tomando y que denotaremos por $H_f$ será el siguiente:

**Definición 5.5.1.** *Sea $f$ un predicado unario cualquiera, definimos el conjunto $H_f$ como*

$$H_f = \{(P, x) \in \tau_C^2 | P \text{ es un PL-programa y } f(x) \in M(P)\} \tag{5.1}$$

De esta manera, $H_f$ estará formado por PL-programas $P$ y términos $x$ tales que $f(x)$ pertenece a la semántica de $P$.

Veamos ahora que $H_f$ es PL-recursivo. Para demostrarlo trataremos de adaptar la demostración clásica de la no recursividad de $H$ al ámbito de la PL-computabilidad. Como veremos, muchas de las ideas de la demostración original van a seguir siéndonos útiles. Esto nos muestra que el tratar de afrontar este problema trasladando conjuntos no recursivos no solo nos ha sido útil por darnos ejemplos que podrían ser no PL-recursivos, sino que también nos ha ayudado de cara a demostrarlo.

**Teorema 5.5.1.** *El conjunto $H_f$ no es PL-recursivo.*

*Demostración.* En caso de $H_f$ fuera PL-recursivo, entonces existiría un PL-programa $P_H$ que computara su función característica, a la que vamos a denotar por $carH$. Veamos que esto nos lleva a un absurdo.

Comenzamos aprovechando la definición que hemos tomado de $H_f$ para razonar acerca de cómo va a estar definida su función característica. Se verificará:

$$carH(P, x) = \begin{cases} suc(cero), & \text{si} \quad f(x) \in M(P) \\ cero, & \text{si} \quad f(x) \notin M(P) \text{ o P no es un PL-programa} \end{cases}$$

Está claro que $carH$ es una función de $\mathcal{T}_C^2 \longrightarrow \mathcal{T}_C$. Vamos a ver que no se va a corresponder con ninguna de las funciones PL-computables entre estos conjuntos. Para ello comprobaremos que no va a poder coincidir con ninguna de ellas a través de un argumento diagonal.

Sea $g$ función PL-computable de $\mathcal{T}_C^2 \longrightarrow \mathcal{T}_C$ cualquiera, veamos que no va a coincidir con $carH$. Para ello trataremos de desarrollar un PL-programa $P$ que verifique para todo $x$

$$\begin{cases} f(x) \in M(P), & \text{si} \quad g(x,x) = cero \\ f(x) \notin M(P), & \text{en} \quad \text{otro caso} \end{cases}$$

Veamos si podemos construirlo. Por ser $g$ PL-computable existirá un PL-programa $P_g$ que la compute, tomamos el PL-programa $P = P_g \cup \{R\}$ siendo $R$ la siguiente cláusula:

```
R: f(X) :- g(X,X,cero).
```

Veamos que efectivamente $P$ computa el predicado $f$ que queremos. Haremos un pequeño abuso del lenguaje en esta demostración utilizando $g$ para referirnos a la función y $g_P$ al predicado del programa.

Como $P$ computa la función $g$ tenemos que $g(x,x) = cero$ si y solo si $g_P(x,x,cero) \in M(P)$.

Dado que $R$ es la única cláusula de $P$ de cuya cabeza $f(x)$ puede ser instancia, tenemos que $g_P(x,x,cero) \in M(P) \Leftrightarrow f(x) \in M(P)$. De este modo, hemos alcanzado que $g(x,x) = 0 \Leftrightarrow f(x) \in M(P)$.

No habrá más elementos de la forma $f(x)$ en $M(P)$ para $x \in \mathcal{T}_C$ ya que $R$ es de la única cláusula de la que podrían deducirse y tan solo genera los $f(x)$ que hemos tratado antes. De esta manera, $f$ queda indefinido en el programa en el resto de los casos, como queríamos.

Acabamos de desarrollar el PL-programa $P$ que buscábamos. Vamos a utilizarlo ahora para demostrar que $g$ y $carH$ no pueden coincidir. Para ello utilizaremos un argumento diagonal.

Como $P$ es un PL-programa, no existe ningún problema en que estudiemos si forma parte de $H_f$ para alguna entrada. En concreto, nos interesará conocer el resultado de $carH(P,P)$. Por otro lado, también podemos estudiar el resultado de aplicar $g(P,P)$ dado que $P$ forma parte de $\mathcal{T}_C$. Si estos valores no coinciden podremos asegurar que $carH$ y $g$ son distintas. Vamos a ello.

1. Si $g(P,P) = cero$ entonces tenemos que $f(P) \in M(P)$.

   Esto nos lleva por definición de $H_f$ a que, como $f(P) \in M(P)$, entonces $carH(P,P) = suc(cero)$.

   De este modo, $f$ y $carH$ no pueden coincidir si $g(P,P) = cero$.

2. Si $g(P,P) \neq cero$ entonces $f(P)$ no forma parte de la semántica de $P$.

   Por lo tanto, como $f(P) \notin M(P)$, tenemos que $carH(P,P) = cero$.

   Vemos como tampoco pueden coincidir en este caso.

Hemos demostrado que $g$ y $carH$ no pueden ser la misma función en ninguno de los dos casos. De este modo, $carH$ no coincide con ninguna función PL-computable, lo que contradice la suposición de que $carH$ era PL-computable. Hemos llegado así al absurdo que buscábamos, por lo que $H_f$ no puede ser PL-recursivo.

□

Acabamos de encontrar un primer ejemplo de conjunto no PL-recursivo. Observamos que para demostrarlo hemos tenido que probar que ninguna de las funciones PL-computables es la que computa su función característica. De este modo, no nos ha valido con no ser capaces de encontrar un programa que computa su función característica, sino que hemos tenido que ir un paso más allá demostrando que no existe.

Recordando la definición de la PL-recusividad de un conjunto como que tanto él como su complementario fueran PL-recursivamente enumerables, podemos asegurar que o $H_f$ o $\overline{H_f}$ no son PL-recursivamente enumerables (podrían no serlo ambos).

Vamos a estudiar ahora cuáles de estos conjuntos lo serán. Para ello aprovecharemos que sabemos que al menos uno de ellos no lo es, con lo que demostrando que $H_f$ es PL-recursivamente enumerable

llegaríamos a que $\overline{H_f}$ no puede serlo.

Está claro que demostrar que un conjunto es PL-recursivamente enumerable va a resultar, en general, mucho más sencillo que probar que no lo es. Esto se debe a que tan solo tendremos que encontrar un PL-programa que compute el predicado *en*, mientras que en el otro caso deberíamos demostrar que ningún PL-programa va a hacerlo.

Para $H_f$ va a ser muy sencillo encontrar este programa. Hemos desarrollado anteriormente un PL-programa universal que nos permitía reconocer si un término formaba parte de la semántica de un programa. Ahora tan solo tendremos que adaptarlo levemente para asegurar que computa el predicado *en* que necesitamos para probar que $H_f$ es PL-computable.

**Proposición 5.5.1.** *El conjunto $H_f$ es PL-recursivamente enumerable.*

*Demostración.* Veamos que podemos desarrollar un PL-programa $P_H$ tal que $en(P,x) \in M(P_H) \Leftrightarrow (P,x) \in H_f$.

Para ello nos va a ser útil nuestro PL-programa universal $U$. Este programa nos permitía dado un término que representaba un PL-programa reconocer que términos forman parte de la semántica de dicho PL-programa.

Sin embargo, no nos basta únicamente con este PL-programa, sino que para poder aplicar el teorema 4.2.1 necesitamos asegurarnos previamente de que el término $P$ que estamos estudiando representa un PL-programa. Un PL-programa que hace esto es el desarrollado en el anexo C, que llamaremos $V$ al igual que hicimos en dicho anexo.

Tomamos el PL-programa $P_H = U \cup V \cup \{R\}$ siendo $U$ nuestro PL-programa universal y $R$ la cláusula:

```
R: enH(P,X) :-
      esPrograma(P),
      solve(const(f,lista(X,vacio)),P).
```

Veamos que efectivamente $P_H$ nos permite demostrar que $H_f$ es PL-recursivamente enumerable.

En primer lugar, por la definición de $H_f$ tenemos que se cumplirá que para dos términos cualesquiera $P$ y $x$, $(P,x) \in H_f$ si y solo si $P$ representa un PL-programa y $f(x) \in M(P)$.

Necesitamos en primer lugar que comprobar que $P$ representa a un PL-programa. Como $P_H$ incluye el PL-programa $V$, entonces verificará para todo término $P$ que $P$ representa un PL-programa $\Leftrightarrow$ $esPrograma(P) \in M(P_H)$.

Ahora bien, en caso de que $P$ represente un PL-programa, por el teorema 4.2.1 para cualquier término ground $x$ tenemos que $f(x) \in M(P) \Leftrightarrow solve(const(f, lista(x, vacio)), P) \in M(P_{H_f})$.

De esta manera, dados dos términos cualesquiera $P$ y $x$, se verifica que $P$ es PL-programa y $f(x) \in M(P)$ si y solo si $esPrograma(P), solve(objetivo(f, lista(x, vacio)), P) \in M(P_H)$.
Dado que $R$ es la única cláusula de $P_{H_f}$ con el predicado $enH$ en su cabeza, llegamos finalmente a que $esPrograma(P)$,$solve(objetivo(f, lista(x, vacio)), P) \in M(P_H)$ si y solo si $enH(P,x) \in M(P_H)$.

Acabamos de demostrar que $enH(P,x) \in M(P_H) \Leftrightarrow (P,x) \in H_f$, por lo que $P_H$ computa el predicado $enH$ que necesitábamos para probar que $H_f$ es PL-recursivamente enumerable.

□

**Corolario 5.5.1.** *El conjunto $\overline{H_f}$ no es PL-recursivamente enumerable.*

*Demostración.* Es concuencia directa del hecho de que $H_f$ sea PL-recursivamente enumerable pero no PL-recursivo.

□

Hemos conseguido el principal objetivo de este capítulo: encontrar un ejemplo de conjunto no PL-recursivo y no PL-recursivamente enumerable. Como veremos, este ejemplo no solo va a ser interesante por sí mismo, sino que podremos utilizarlo para encontrar otros muchos ejemplos de conjuntos no PL-recursivos. Veremos este desarrollo en el siguiente capítulo.

Nos gustaría antes de cerrar este capítulo incluir una breve reflexión acerca del predicado *not* y cómo nos va a resultar imposible de computar con nuestros PL-programas. El interés de este resultado reside en que nos ayuda a comprender la enorme importancia que tiene el no poder computarlo, que es lo que nos lleva a que existan conjuntos no PL-recursivos pero sí PL-recursivamente enumerables.

Antes de pasar a desarrollar estos resultados nos gustaría dejar claro a que nos estamos refiriendo al hablar de este predicado *not*. La idea intuitiva es que se trata de un predicado que nos indica si algo no se verifica.

Vamos a trasladarlo al ámbito de los PL-programas como un predicado que cumpla para todo PL-programa $P'$ y término $x$ que $not(P', x) \in M(P) \Leftrightarrow x \notin M(P')$. De esta manera *not* nos permitiría reconocer cuando un término no forma parte de la semántica de un PL-programa.

**Teorema 5.5.2.** *No podemos generar un PL-programa que compute el predicado not. Es decir, no existe un PL-programa $P$ tal que para cada PL-programa cualquiera $P'$ y término $x$ se verifique $not(P', x) \in M(P) \Leftrightarrow x \notin M(P')$.*

*Demostración.* En caso de que existiera tal PL-programa $P$, vamos a demostrar que todo conjunto PL-recursivamente enumerable sería PL-recursivo, por lo que estas definiciones pasarían a ser equivalentes. Esto nos llevaría a un absurdo, ya que acabamos de encontrar en $H_f$ un ejemplo de conjunto PL-recursivamente enumerable pero no PL-recursivo. Por lo tanto, nos basta con demostrar que si tal $P$ existiera entonces todo conjunto PL-recursivamente enumerable sería PL-recursivo. Vamos a ello.

Sea $D$ conjunto PL-recursivamente enumerable cualquiera, tomamos $P_D$ PL-programa que computa el predicado *enD*. Vamos a demostrar que $D$ va a ser PL-recursivo, para lo que nos basta con demostrar que su complementario también será PL-recursivamente enumerable.

Tomamos el PL-programa $P_{\overline{D}} = P \cup \{R\}$ siendo $R$ la cláusula:

`R: enNoD(`$X_1$`,..., `$X_n$`) :- not(P, enD(`$X_1$`,..., `$X_n$`)).`

Veamos que el predicado *enNoD* de $P_{\overline{D}}$ nos vale para probar que $\overline{D}$ es PL-recursivamente enumerable.

Tenemos que para todo $\overline{x} = (x_1, \ldots, x_n) \in \mathcal{T}_C$ se cumplirá que $\overline{x} \in \overline{D}$ si y solo si se tiene que $enD(\overline{x}) \notin M(P_{\overline{D}})$ por computar este programa el predicado *enD*.

Ahora, por definición de $P_{\overline{D}}$ como computa el predicado *not*, tendremos que $enD(\overline{x}) \notin M(P) \Leftrightarrow not(P, enD(\overline{x})) \in M(P_{\overline{D}})$.

Pero al ser $R$ la única cláusula con *enNoD* en su cabeza, tenemos que finalmente $not(P, enD(\overline{x})) \in M(P_{\overline{D}})$ si y solo si $enNoD(\overline{x}) \in M(P_{\overline{D}})$.

Hemos llegado a que $\overline{x} \in \overline{D} \Leftrightarrow enNoD(\overline{x}) \in M(P_{\overline{D}})$, lo que nos lleva a que $\overline{D}$ es PL-recursivamente enumerable. De esta manera, $D$ sería PL-recursivo.

Tenemos, por tanto, que en caso de que tal $P$ existiera los conjuntos PL-recursivamente enumerables serían también PL-recursivos. Como hemos visto, $\overline{H_f}$ es PL-recursivamente enumerable pero no PL-recursivo, con lo que hemos llegado a una contradicción

□

Hemos visto que mediante nuestro programa universal nos es sencillo estudiar qué términos forman parte de $M(P)$, pero estudiar los que no están en él nos va a ser imposible.

Ahora que ya hemos encontrado un primer ejemplo de conjunto no PL-recursivo nos podemos preguntar si vamos a ser capaces de encontrar más. Demostrar que $H_f$ no es PL-recursivo no ha sido sencillo y hemos tenido que recurrir a un argumento diagonal. ¿Tendremos que realizar un razonamiento similar cada vez que queramos encontrar un conjunto no PL-recursivo?

Vamos a dedicar el siguiente capítulo a contestar esta pregunta, viendo que no siempre va a ser así. Estudiaremos cómo mediante la noción de reducibilidad vamos a ser capaces de hacer depender la PL-recursividad de conjuntos de la de otros que sí conozcamos.

De este modo, saber que $H_f$ no es PL-recursivo no solo nos va a dar información sobre el propio $H_f$, sino que nos va a permitir encontrar otros muchos conjuntos no PL-recursivos. Más aún, veremos cómo teoremas fundamentales de la teoría de la PL-computabilidad como el Teorema de Rice van a basar su demostración en lo que ya sabemos de $H_f$, sin tener que recurrir a ningún argumento diagonal.

La idea que vamos a seguir se asemeja en parte a lo que hemos hecho en este último teorema. Supondremos que el conjunto que queremos estudiar es PL-recursivo y veremos que esto nos lleva a una contradicción con el hecho de que $H_f$ no lo es.

Además, desarrollaremos una técnica conocida como PL-reducción que va a faciliar enormemente esta tarea, como veremos en el siguiente capítulo.

# Capítulo 6

# Reducibilidad

Ahora que hemos encontrado un primer conjunto no PL-recursivo vamos a tratar de utlizarlo para encontrar más.

A priori, como vimos en la demostración de que $H_f$ no es PL-recursivo, el probar que un conjunto no es PL-recursivo pasa por demostrar qué ninguna de las funciones PL-computables coincide con su función característica. Esto no va a ser siempre sencillo, por lo que resultaría de interés encontrar alguna manera más directa de alcanzar este resultado.

Esta dificultad también aparece en el ámbito de la Turing-computabilidad al demostrar que un conjunto no es recursivo. En ese caso, a veces se puede resolver haciendo depender la recursividad de un conjunto de la de otro que sí conocemos. Para ello, en el caso de las máquinas de Turing solíamos comprobar que si existiera una máquina que nos dijera si un elemento forma parte del segundo conjunto o no, entonces podríamos adaptarla para que nos lo dijera del primero.

La idea detrás de esta técnica se conoce como reducción y no solo va a ser útil en el ámbito de la Turing-computabilidad, sino que se trata de una de las principales técnicas matemáticas para resolver problemas (un caso particular de ella son las reducciones al absurdo). De modo general, una reducción consiste en hacer depender un primer problema de un segundo, de forma que si encontramos una manera de solucionar el segundo problema podríamos usarla para resolver el primero. En un sentido informal, estaríamos diciendo que el primer problema es más sencillo que el segundo, porque ser capaces de resolver este segundo problema nos garantiza poder resolver el primero.

Esto también nos va a resultar útil de cara a estudiar los conjuntos no PL-recursivos. El esquema que vamos a seguir es similar al clásico. Trataremos de reducir el problema de pertenecer a un conjunto $B$ al de pertenecer a otro conjunto $A$. Entonces en caso de que $A$ fuera PL-recursivo tendríamos que $B$ también lo sería.

En nuestro caso estamos interesados en probar que un conjunto no es PL-recursivo, que es justo lo contrario de lo que nos aporta la reducción. Por este motivo, nosotros las utilizaremos para encontrar contradicciones con lo que ya conocemos acerca de conjuntos como $H_f$. Sabemos que $H_f$ no es PL-recursivo, por lo que si conseguimos reducir $H_f$ a otro conjunto $A$, tendremos que $A$ tampoco podrá ser PL-recursivo, ya que si lo fuera llegaríamos a que también $H_f$ lo sería, lo que es absurdo.

Esta técnica es también aplicable al caso de la PL-enumerabilidad recursiva, no solo de la PL-recursividad, pues también tendremos que si un conjunto $A$ se reduce a otro $B$ que es PL-recursivamente enumerable entonces $A$ también lo sería. Por tanto, si $A$ no es PL-recursivamente enumerable y $A$ se reduce a $B$, concluiremos que $B$ no es PL-recursivamente enumerable.

Una vez que hayamos visto algunos ejemplos de conjuntos PL-recursivos vamos a pasar a desarrollar un tipo concreto de reducciones conocidas como PL-reducciones. Las PL-reducciones nos van a permitir en algunos casos simplificar la demostración de que un conjunto no es PL-recursivo o PL-recursivamente enumerable. De este modo, desarrollaremos una serie de demostraciones y resultados

relativos a ellas y veremos como algunas de las demostraciones que habíamos desarrollado previamente mediante reducciones van a simplificarse.

## 6.1. Ejemplos de reducciones y conjuntos no PL-recursivos

Vamos a ver en esta sección algunos ejemplos de conjuntos no PL-recursivos. Muchos de ellos van a tratarse de ejemplos clásicos de conjuntos no recursivos adaptados al ámbito de los PL-programas. El esquema que seguiremos va a ser similar en todos ellos: en primer lugar definiremos el conjunto, para pasar después a probar que no es PL-recursivo.

Todas las demostraciones van a estar basadas en una misma técnica que consistirá en reducir a este nuevo conjunto otro del que ya hayamos demostrado que no es PL-recursivo. De este modo, suponer que el conjunto que estamos estudiando es PL-recursivo nos llevaría a que este segundo conjunto también lo sería, lo que es un absurdo.

Todos los ejemplos que vamos a ver se refieren a conjuntos de PL-programas. Para definirlos lo que haremos es establecer una condición sobre su semántica y tomar el conjunto de PL-programas que la cumplan. De este modo, por ejemplo, hablaremos de los PL-programas que no aceptan nunca un predicado o de los PL-programas que son semánticamente equivalentes.

### 6.1.1. No PL-recursividad de $E_f$

Comenzamos definiendo un nuevo conjunto del que vamos a demostrar que no es PL-recursivo. Se va a tratar del conjunto formado por los PL-programas que en uno de sus predicados $f$ cumplen $f(x) \notin M(P)$ para cualquier término $x \in \mathcal{T}_C$. Es decir, aquellos en los que el predicado $f$ no se cumple para ningún argumento, o sea, tiene semántica vacía.

**Definición 6.1.1.** *El conjunto $E_f$ es el conjunto de PL-programas definido por*

$$E_f = \{P | P \text{ es PL-programa y } \nexists x \in \tau_C \text{ tal que } f(x) \in M(P)\} \tag{6.1}$$

Vamos a demostrar ahora que este conjunto no es PL-recursivo. Para ello, en vez de recurrir a argumentos diagonales como hicimos anteriormente, vamos a tratar de aprovechar lo que ya conocemos acerca de conjuntos no PL-recursivos.

Sabemos que $H_g$, definido en 5.5.1, no es PL-recursivo. Si somos capaces de hacer depender la PL-recursividad de $H_g$ de la de $E_f$, tendremos que $E_f$ tampoco puede ser PL-recursivo.

Hacer esta reducción no nos va a resultar difícil dado que $H_g$ y $E_f$ están bastante relacionados. El problema de decisión en $H_g$ para un programa $P$ consiste en saber si un término $g(x)$ forma parte de su semántica . Ahora lo que queremos saber es si uno de sus predicados $f$ jamás va a formar parte de la semántica.

Como veremos en la demostración, lo que haremos para reducir $H_g$ a $E_f$ es construir un PL-programa tal que $f$ tan solo se satisfaga si $g(x) \in M(P)$. De este modo, tendremos que $g(x) \in M(P)$ si y solo si $f$ se satisface en algún término.

Procedemos a realizar de manera formal este razonamiento.

**Teorema 6.1.1.** *El conjunto $E_f$ no es PL-recursivo.*

*Demostración.* Veamos que podemos reducir el problema de decisión de $H_g$ al de $E_f$, de modo que si $E_f$ fuera PL-recursivo entonces $H_g$ también lo sería.

Si $E_f$ fuera PL-recursivo tendríamos que existiría un PL-programa que computara su función característica $carE_f$ que quedaría definida por:

$$carE_f(P) = \begin{cases} suc(cero), & \text{si} \quad \nexists x \in \mathcal{T}_C : f(x) \in M(P) \\ cero, & \text{si} \quad \exists x \in \mathcal{T}_C : f(x) \in M(P) \end{cases}$$

Denotaremos tal PL-programa por $P_E$.

Como hemos anunciado, llegaremos a nuestra contradicción desarrollando un PL-programa que compute la función característica de $H_g$. Para ello tomamos el PL-programa $P_H = P_E \cup \{R_1, R_2, R_3\}$ siendo estas las cláusulas siguientes:

```
R1: convierte(P,X,lista(regla(const(f,lista(X,vacio)),lista(const(g,lista(X,vacio)),
        lista(const(g,lista(X,vacio)),vacio)),P)).

R2: carH(P,X,cero) :-
      convierte(P,X,P1),
      carEf(P1, suc(cero)).

R3: carH(P,X,suc(cero)) :-
      convierte(P,X,P1),
      carEf(P1, cero).
```

Veamos que este programa computa $carH$, lo que como hemos dicho nos lleva a una contradicción.

En primer lugar, la cláusula *convierte* se encarga de dado un PL-programa $P$ y un término $x$ aceptar únicamente el programa que sea idéntico a $P$ añadiéndole en el comienzo la cláusula $R$:

```
R: f(x) :- g(x).
```

Vemos como siempre va a existir un $P_1$ tal que $convierte(P, x, P_1) \in M(P_H)$ (nos basta con añadir esta cláusula al comienzo de la lista de cláusulas del programa $P$). Además, resulta sencillo comprobar que $f(x') \in M(P_1) \Leftrightarrow x' = x$ y $g(x) \in M(P)$ ya que es la única cláusula de $P_1$ con el predicado $f$ en su cabeza.

Dado un PL-programa $P$ y un término $x$ pueden darse dos casos según $g(x)$ pertenezca a $M(P)$ o no. Estudiamos ahora ambos para comprobar que $R$ efectivamente computa $carH$.

1. Estudiamos ahora el caso de $g(x) \in M(P)$. Vamos a demostrar que $g(x) \in M(P) \Leftrightarrow carH(P, x, suc(cero)) \in M(P_H)$.

   Comenzamos observando que si $g(x) \in M(P)$ entonces $f(x) \in M(P_1)$ como hemos visto.

   Está claro que al cumplirse que $f(x) \in M(P_1)$, entonces $P_1$ no forma parte de $E_f$. De este modo, como $P_H$ computa $carE_f$, tenemos que $carE_f(P_1, cero) \in M(P_H)$.

   Por último, a través de la cláusula $R_3$ llegamos a que, como $convierte(P, x, P_1), carE_f(P_1, cero) \in M(P_H)$, entonces $carH(P, x, suc(cero)) \in M(P_H)$. Hemos llegado así a que $g(x) \in M(P) \Rightarrow carH(P, x, suc(cero)) \in M(P_H)$.

   Para el otro sentido de la implicación, tenemos que $carH(P, x, suc(cero)) \in M(P_H)$ si y solo si $\exists P'$ tal que $convierte(P, x, P'), carE_f(P', cero) \in M(P_H)$.

   Como vimos, el único $P'$ que podría verificarlo sería el que se construye añadiendo al comienzo de $P$ la cláusula $R$.

   De este modo llegamos a que, como dicha $R$ será la única cláusula de $P'$ con $f$ en su cabeza, tendremos que $g(x) \in M(P_1)$. Evidentemente, como tan solo hemos añadido a $P_1$ la cláusula $R$ respecto a $P$ que no afecta al resto de la semántica, llegamos a que $g(x)$ también pertenecerá a la semántica de $P$.

   Acabamos de demostrar que $carH(P, x, suc(cero)) \in M(P_H) \Rightarrow g(x) \in M(P)$ con lo que ya

tenemos demostradas las dos implicaciones para este caso.

2. Vamos ahora a demostrar que $g(x) \notin M(P) \Leftrightarrow carH(P, x, cero) \in M(P_H)$. Al igual que en el caso anterior estudiamos una a una las dos implicaciones.

   Comenzaremos probando que $g(x) \notin M(P) \Rightarrow carH(P, x, cero) \in M(P_H)$.

   Si $g(x) \notin M(P)$ entonces no existe un término $x'$ tal que $f(x') \in M(P_1)$ al ser $R$ la única cláusula de $P_1$ que contiene a $f$ en su cabeza.

   Por lo tanto, como $f$ nunca se satisface en $P_1$ tenemos que $carE_f(P_1, suc(cero)) \in M(P_H)$.

   Aplicando $R_2$ obtenemos que $carH(x, P, cero) \in M(P_H)$. De este modo, acabamos de demostrar la primera implicación.

   Para el otro sentido, tenemos que si $carH(P, x, cero) \in M(P_H)$ entonces, como $R_2$ es la única cláusula de $P_H$ de la que puede ser instancia, tenemos que existe un $P'$ tal que $convierte(P, x, P')$, $carE_f(P', suc(cero)) \in M(P_H)$.

   $P'$ tan solo puede ser el PL-programa que obtenemos añadiendo a $P$ la cláusula $R$ y denotamos por $P_1$, ya que es el único que verifica que $convierte(P, x, P') \in M(P_H)$.

   Tenemos, por tanto, que $carE_f(P_1, suc(cero)) \in M(P_H)$. De este modo, se verifica que $f(x) \notin M(P_1)$, lo que implica que $g(x) \notin M(P)$.

Hemos llegado a que la función $carH$ computada por $P_H$ quedaría definida como:

$$h(P, x) = \begin{cases} suc(cero), & \text{si} \quad g(x) \in M(P) \\ cero, & \text{si} \quad g(x) \notin M(P) \end{cases}$$

con lo que se correspondería con la función característica de $H_g$. Esto es absurdo, hemos probado que $H_g$ es no PL-recursivo pero acabamos de encontrar un programa que computa su función característica.

Por lo tanto, acabamos de demostrar que suponer que $E_f$ es PL-recursivo implica que $H_g$ también lo es, con lo que concluimos que $E_f$ no puede ser PL-recursivo.

□

Hemos demostrado que el conjunto $E_f$ no es PL-recursivo. Este resultado es importante, ya que nos muestra que no podemos computar una función que nos diga cuando un predicado nunca va a satisfacerse. Esto no nos debería extrañar, ya que en cierto modo se corresponde con la traducción al ámbito de la PL-programación del conjunto formado por las máquinas de Turing que no paran para ninguna entrada. Este conjunto no es Turing-recursivo, por lo que no es ninguna sorpresa que su equivalente no sea PL-recursivo.

Por último, nos gustaría resaltar el hecho de que que $E_f$ sea no PL-recursivo implica que o $E_f$ o $\overline{E_f}$ no van a ser PL-recursivamente enumerables.

Acabamos de ver en esta sección un primer ejemplo de conjunto no PL-recursivo obtenido a través de reducciones. La importancia de este ejemplo radica en que nos muestra de manera detallada el esquema que van a seguir nuestras demostraciones por reducción. Además, nos aporta un nuevo conjunto sobre el que poder reducir otros, como haremos ahora.

### 6.1.2. No PL-recursividad de $IG_f$

Vamos ahora con un nuevo conjunto del que vamos a demostrar que no es PL-recursivo. Se trata del formado por las tuplas de PL-programas que tienen la misma semántica para uno de sus predicados $f$.

Comenzamos al igual que en el caso anterior presentando la definición formal de este conjunto.

**Definición 6.1.2.** *El conjunto $IG_f$ es el conjunto de pares de PL-programas definido por*

$$IG_f = \{(P_1, P_2) | P_1, P_2 \text{ son PL-programas cumpliendo } \forall x \in \tau_C \text{ que } f(x) \in M(P_1) \Leftrightarrow f(x) \in M(P_2)\} \tag{6.2}$$

En este caso la reducción que vamos a hacer es mucho más sencilla que la de la demostración anterior.

Vamos a ver como en caso de que $IG_f$ fuera PL-recursivo entonces podríamos construir un programa que computara la función característica de $E_f$. Para ello, dado un PL-programa $P$ lo que haremos es compararlo con uno que sepamos que no se satisface nunca para el predicado $f$, como puede ser el programa vacío. Si los programas tienen la misma semántica en $f$ tendremos que $P$ no se satisfacería nunca por lo que pertenecería a $E_f$. Si sus semánticas no coinciden en $f$ entonces tendremos que $f$ se satisface en $P$ en al menos un término, por lo que no pertenecería a $E_f$.

De esta manera, si $IG_f$ es PL-recursivo, entonces $E_f$ también lo sería. Esto es absurdo porque hemos probado que $E_f$ no es PL-recursivo.
Pasamos a desarrollar formalmente estas ideas.

**Teorema 6.1.2.** *El conjunto $IG_f$ no es PL-recursivo.*

*Demostración.* Vamos a demostrar que si $IG_f$ fuera PL-recursivo entonces $E_f$ también lo sería.

Comenzamos observando que si $IG_f$ es PL-recursivo, entonces existe un PL-programa $P_{IG}$ que computa su función característica $carIG_f$ que está definida como:

$$carIG_f((P_1, P_2)) = \begin{cases} suc(cero), & \text{si} \quad \forall x \in \mathcal{T}_C \ f(x) \in M(P_1) \Leftrightarrow f(x) \in M(P_2) \\ cero, & \text{en caso contrario} \end{cases}$$

Veamos que aprovechando $P_{IG}$ vamos a ser capaces de construir un PL-programa que compute $carE_f$, lo que nos llevaría a un absurdo. Tomamos $P_E = P_{IG} \cup \{R_1, R_2\}$ siendo $R_1, R_2$ las siguientes cláusulas:

```
R1: programaVacio(vacio).

R2: carEf(P,Y) :-
      programaVacio(P1),
      carIGf(P,P1, Y).
```

Veamos que $P_E$ computará $carE_f$.

En primer lugar, al ser $R_1$ la única cláusula de $P_E$ con el predicado $programaVacio$ en su cabeza tenemos $programaVacio(P_v) \in M(P_E) \Leftrightarrow P_v = vacio$. De este modo, tenemos que si $programaVacio(P_v) \in M(P_E)$ entonces $M(P_v) = \emptyset$, dado que se trata de un programa sin ninguna cláusula.

Es sencillo observar que las semánticas de $P_v$ y un PL-programa $P$ coincidirán en el predicado $f$ si y solo si $f(x) \notin M(P)$.

Estudiemos ahora detalladamente cómo actúa $R_2$ sobre la semántica de nuestro programa. Dado un PL-programa cualquiera $P$ tenemos:

1. Si para todo $x$ se cumple que $f(x) \notin M(P)$ entonces está claro que su semántica coincide con la de $P_v$ como antes dijimos. Por lo tanto, en este caso se verificará que no existe un $x$ tal que $f(x) \in M(P)$ si y solo si $carIG_f(P, P_v, suc(cero)) \in M(P_E)$.

   Por otro lado, por ser $R_2$ la única cláusula de $P_E$ con $carE_f$ en su cabeza y $P_v$ el único término que verifica $programaVacio$, tenemos que $carE_f(P, suc(cero)) \in M(P_E)$ si y solo si $programaVacio(P_v), carIG(P, P_v, suc(cero)) \in M(P_E))$.

Si unimos ambas expresiones llegamos a que $carE(P, suc(cero)) \in M(P_E) \Leftrightarrow \nexists x$ tal que $f(x) \in M(P)$.

2. Si existe un $x \in \mathcal{T}_C$ tal que $f(x) \in M(P)$ entonces las semánticas de $P_v$ y $P$ no coincidirán en $f$, por lo que $(P, P_v) \notin IG_f$.

   Dado que $P_E$ computa $carIG_f$ entonces se verificará que $carIG_f(P_1, P_2, cero) \in M(P_E) \Leftrightarrow \exists x \in \mathcal{T}_C : f(x) \in M(P)$.

   Con un razonamiento similar al punto anterior, por $R_2$ llegamos a que $carE_f(P, cero) \in M(P_E) \Leftrightarrow \exists x \in \mathcal{T}_C$ tal que $f(x) \in M(P)$.

Nuestro PL-programa $P_E$ computará, por lo tanto, la función $carE_f$ que como hemos visto queda definida por:

$$carE_f(P) = \begin{cases} 1, & \text{si} \quad \nexists x \in \mathcal{T}_C : f(x) \in M(P) \\ 0, & \text{si} \quad \exists x \in \mathcal{T}_C : f(x) \in M(P) \end{cases}$$

Evidentemente esta función se corresponde con la función característica de $E_f$. Hemos llegado, por tanto, a que $E_f$ es PL-recursivo, lo que sabemos que es falso. Esta contradicción se debe al hecho de suponer que $IG_f$ es PL-recursivo.

□

Acabamos de encontrar un nuevo ejemplo de conjunto no PL-recursivo. Más allá de haber encontrado este ejemplo, queremos resaltar como esta demostración y la anterior siguen un esquema similar. Esto nos puede hacer reflexionar acerca de si sería posible generalizar algunos de los pasos que hemos dado, porque parece que van a ser comunes a todas las reducciones.

Para ello definiremos un tipo de reducción muy utilizada, que llamaremos PL-reducción, y demostraremos que efectivamente nos va a permitir probar la no PL-recursividad y no PL-recursividad enumerable de conjuntos. Prestaremos especial atención en observar cómo su uso simplificaría alguno de los ejemplos que hemos desarrollado para dejar clara la utilidad de este tipo de reducciones.

La motivación para buscar esta generalización la encontramos al observar nuestras últimas demostraciones. Todas siguen un esquema muy similar, variando únicamente el PL-programa que usan para relacionar las funciones características de los conjuntos. Esto nos lleva a tratar de generalizar muchos de los pasos que hemos seguido para no tener que repetir todo el esquema en cada demostración, sino tan solo concentrarnos en las partes que van a ser distintas según el conjunto que estemos estudiando.

Aunque todos los resultados que podemos alcanzar mediante PL-reducciones también los podemos obtener a través de reducciones como las que hemos visto en los ejemplos, vamos a comprobar cómo el uso de PL-reducciones simplifica las demostraciones.

## 6.2. PL-Reducibilidad

Como hemos comentado anteriormente, vamos a tratar de desarrollar en esta sección una serie de definiciones y teoremas que nos permitan trabajar con reducciones de un modo más sencillo y general. El mecanismo que usaremos va a ser igual al seguido en los ejemplos de la sección anterior, pero nos va a permitir generalizar alguno de los pasos que hemos dado, de manera que no vamos a tener que repetirlos en cada demostración que hagamos.

Para ello vamos a desarrollar un tipo de reducción específico de este tipo de problemas que conoceremos como PL-reducción. Con él trataremos de expresar cómo podemos relacionar entre si dos conjuntos mediante una función PL-computable. De este modo, buscaremos una función $g$ que nos permita estudiar si un elemento $x$ pertenece a un conjunto según $g(x)$ pertenezca a otro.

Definamos por tanto formalmente este concepto de PL-reducción y veamos un ejemplo de ella antes de pasar a estudiar la relación que van a tener las PL-reducciones con la PL-reducibilidad de conjuntos. Para esta definición nos hemos basado en la de Turing-reducción de (Sipser, 2012), aunque la hemos adaptado a nuestro modelo de programas lógicos.

**Definición 6.2.1** (PL-reducción)**.** *Decimos que un conjunto $A \subseteq \mathcal{T}_C^n$ es PL-reducible a otro conjunto $B \subseteq \mathcal{T}_C^m$, y lo escribiremos como $A \leq_{PL} B$, si existe una función $g$ PL-computable y total tal que para cada $\overline{x} = (x_1, \ldots, x_n) \in \mathcal{T}_C^n$ verifique:*

$$\overline{x} \in A \Leftrightarrow g(\overline{x}) \in B \tag{6.3}$$

*Llamaremos a esta función PL-reducción de $A$ en $B$.*

Una PL-reducción de $A$ en $B$ nos proporciona una manera de estudiar la pertenencia a $A$ a través de lo que ocurre en $B$. De este modo, si queremos saber si $\overline{x} \in A$ nos basta con estudiar si $g(\overline{x}) \in B$.

Veamos un ejemplo de PL-reducción que posteriormente nos va a resultar útil.

**Proposición 6.2.1.** *Los conjuntos $E_f$ y $IG_f$, definidos en 6.1.1 y 6.1.2 respectivamente, verifican $E_f \leq_{PL} IG_f$. Es decir, $E_f$ es PL-reducible a $IG_f$.*

*Demostración.* Denotamos por $P_{vacio}$ al programa que no está formado por ninguna cláusula, para el que $M(P) = \emptyset$.

Observamos que $P \in E_f \Leftrightarrow$ no existe un $x$ tal que $f(x) \in M(P)$. Ahora bien, dado que no existe un $x$ tal que $f(x) \in M(P_{vacio})$, llegamos a que $P$ no será satisfactible en $f$ si y solo si sus semántica con respecto a $f$ coincide con la del programa vacío, es decir, $P \in E_f \Leftrightarrow (P, P_{vacio}) \in IG_f$.

De este modo, nos basta con probar que la función $g$ que para cada $P$ devuelve la tupla $(P, P_{vacio})$ es PL-computable. Tomamos:

```
g(P,P,lista(vacio)).
```

Este programa computa la función $g$. Ahora bien, como $P \in E_f \Leftrightarrow g(P) \in IG_f$ tenemos que $E_f$ es PL-reducible a $IG_f$. □

Acabamos de ver un ejemplo de PL-reducción entre dos conjuntos. Nos gustaría resaltar la gran semejanza de la PL-reducción que acabamos de hacer con la que seguimos en la demostración del teorema 6.1.2. Esto sucede ya que la idea detrás de las PL-reducciones es la misma que seguíamos en las reducciones de las sección anterior: hacer depender la pertenencia de un conjunto de la de otro.

Antes de pasar a estudiar la relación entre PL-recursividad y PL-reducciones vamos a incluir un sencillo lema que utilizaremos más adelante.

**Lema 6.2.1.** *Se verifica que $A \leq_{PL} B \Leftrightarrow \overline{A} \leq_{PL} \overline{B}$.*

*Demostración.* Es consecuencia directa de la definición de PL-reducción, tenemos que $x \in A \Leftrightarrow g(x) \in B$, por lo que trivialmente $x \notin A \Leftrightarrow g(x) \notin B$. □

Pasamos ahora al resultado fundamental de esta sección. Hemos visto que si un conjunto es PL-reducible a otro entonces estudiar la pertenencia a este segundo conjunto nos permite conocer el primero. Veamos ahora como esto va a tener implicaciones muy importantes en cuanto a la PL-recursividad de estos conjuntos.

El hecho de que si $A \leq_{PL} B$ entonces para saber si $x \in A$ baste conocer si $g(x) \in B$, nos puede hacer sospechar que decidir si un elemento pertenece a $A$ va a ser más fácil que para $B$. Veamos como efectivamente va a ser así en el siguiente teorema. Aunque ahora mismo tan solo vamos a concentrarnos en la PL-recursividad, veremos más adelante como el mismo resultado puede obtenerse para la

PL-recursividad enumerable de conjuntos.

**Teorema 6.2.1.** *Si $A \leq_{PL} B$ y $B$ es PL-recursivo entonces $A$ también lo es.*

*Demostración.* Tenemos que $B$ es PL-recursivo, por lo que existirá un PL-programa $P_B$ que computa su función característica.

Por otro lado, sea $g$ la PL-reducción entre ellos, tomamos el PL-programa $P_g$ que la compute. Vamos a tratar de formar a partir de ellos un PL-programa que compute la función característica de $A$.

Tomamos $P_A = P_B \cup P_g \cup \{R\}$ siendo $R$ la cláusula:

```
R: carA(X1,...,Xn,V) :-
      g(X1,...,Xn,Y1,...,Ym),
      carB(Y1,...,Ym,V).
```

Veamos que $P_A$ computa la función característica de $A$, para lo que estudiaremos los dos casos posibles según $\overline{x} = (x_1, \ldots, x_n)$ pertenezca o no a $A$:

1. Sea $\overline{x} \in A$, entonces $\overline{x} \in A$ si y solo si $g(\overline{x}) \in B$.

   Tenemos que $P_A$ computa $g$, por lo tanto $g(\overline{x}) = \overline{y} \Leftrightarrow g_P(\overline{x}, \overline{y}) \in M(P_A)$. Realizamos aquí un abuso de notación refiriendonos por $g_P$ al predicado computado por el programa y por $g$ a la función.

   Por otro lado, como el predicado $carB$ computa la función característica de $B$ se verifica $carB(\overline{y}, 1) \in M(P_A) \Leftrightarrow \overline{y} \in B$.

   Combinando ambos resultados y partiendo de que $\overline{x} \in A \Leftrightarrow g(\overline{x}) \in B$ llegamos a que esto ocurrirá si y solo si $g_P(\overline{x}, \overline{y}), carB(\overline{y}, suc(cero)) \in M(P_A)$. Ahora bien, como $R$ es la única cláusula con $carA$ en su cabeza llegamos a que esto sucede si y solo si $carA(\overline{x}, suc(cero)) \in M(P_A)$.

   De este modo hemos alcanzado que $\overline{x} \in A \Leftrightarrow carA(\overline{x}, suc(cero)) \in M(P_A)$.

2. Para $\overline{x} \notin A$ el razonamiento es análogo y nos permite concluir que $\overline{x} \notin A \Leftrightarrow carA(\overline{x}, cero) \in M(P_A)$.

Hemos probado que $P_A$ computa la función característica de $A$, por lo que $A$ es PL-recursivo.

□

Antes de explicar más profundamente las implicaciones de este resultado vamos a añadir un sencillo corolario.

No estamos interesados en comprobar si un conjunto es PL-recursivo, sino más bien en encontrar ejemplos de conjuntos que no lo sean. Por este motivo no vamos a utilizar tanto el teorema que acabamos de demostrar sino en su contrarrecíproco, que enunciamos ahora. Al igual que en el teorema anterior, también vamos a poder desarrollar este resultado para la PL-recursividad enumerable de conjuntos, como haremos más adelante.

**Corolario 6.2.1.** *Si $A \leq_{PL} B$ y $A$ no es PL-recursivo, entonces $B$ no es PL-recursivo.*

*Demostración.* Se trata del contrarrecíproco del teorema 6.2.1. □

Este resultado va a ser muy importante para nosotros, ya que nos va a permitir estudiar la PL-recursividad de conjuntos construyendo PL-reducciones a otros conjuntos que conozcamos.

Vemos que esta idea de hacer depender conjuntos de otros que conocemos es exactamente la misma que seguíamos en los ejemplos de la primera sección de este capítulo. Sin embargo, las PL-reducciones nos simplifican esta tarea. El teorema que acabamos de demostrar nos permite probar de manera general la parte común que seguíamos en todas las demostraciones por reducciones. Veamos un ejemplo de ello.

Antes hemos demostrado que $E_f \leq_{PL} IG_f$. Ahora bien, el conjunto $E_f$ es uno que conocemos bien y que ya probamos en 6.1.1 que no es PL-recursivo. Veamos como consecuencia directa del corolario que acabamos de desarrollar que $IG_f$ tampoco va a serlo.

**Corolario 6.2.2.** *El conjunto $IG_f$ es no PL-recursivo.*

*Demostración.* Aplicando el corolario 6.2.1, como $E_f \leq_{PL} IG_f$ y $E_f$ no es PL-recursivo obtenemos que $IG_f$ tampoco lo es. □

Vemos que acabamos de llegar al resultado que antes obtuvimos en el teorema 6.1.2 de una manera mucho más sencilla y sin necesidad de desarrollar un PL-programa tan complicado como el que hicimos en esa demostración.

Este es un ejemplo de cómo el uso de PL-reducciones nos permite trabajar con reducciones sin tener que desarrollar las funciones características de los conjuntos, sino limitándonos a estudiar la relación entre ellos.

Antes de pasar a estudiar la manera en que vamos a poder aplicar las PL-reducciones a estudiar si un conjunto es PL-recursivamente enumerable, vamos a ver un nuevo ejemplo de conjunto no PL-recursivo que podemos encontrar gracias a las PL-reducciones.

### 6.2.1. No PL-recursividad de $FIN_f$

Veamos ahora un nuevo ejemplo de conjunto del que podemos probar que no es PL-recursivo haciendo uso de las PL-reducciones. Se trata del conjunto formado por los PL-programas que aceptan únicamente un número finito de términos para uno de sus predicados $f$.

**Definición 6.2.2.** *El conjunto $FIN_f$ es el conjunto de PL-programas definido por*

$$FIN_f = \{P | P \text{ es un PL-programa y el conjunto } \{x : f(x) \in M(P)\} \text{ es finito}\} \tag{6.4}$$

Estudiaremos este nuevo ejemplo de una manera similar a como lo hemos hecho con $IG_f$. Para ello, en primer lugar encontraremos una PL-reducción entre este conjunto y $E_f$, que no es PL-recursivo.

Aunque en este caso la función $g$ que vamos a construir es bastante más complicada, veremos que la estructura de la demostración no cambia. Buscamos reducir $E_f$ a $FIN_{f'}$, con lo que la función $g$ que pretendemos construir es una que para todo $P$ verifique que $P \in E_f \Leftrightarrow g(P) \in FIN_{f'}$. La idea que usaremos para desarrollar $g$ es la de construir un programa de manera que si para algún $x$, $f(x) \in M(P)$ entonces existan infinitos $x'$ tales que $f'(x') \in M(g(P))$.

**Proposición 6.2.2.** *Los conjuntos $E_f$ y $FIN_{f'}$ verifican $E_f \leq_{PL} FIN_{f'}$. Es decir, $E_f$ es PL-reducible a $FIN_{f'}$.*

*Demostración.* Tomamos $g$ la función que dado un programa $P$ añade en él las cláusulas $R_1$ y $R_2$:

```
R1: f'(X) :- f(X).
R2: f'(repite(X)) :- f'(X).
```

Esta función es evidentemente PL-computable, un PL-programa que la computa sería el siguiente:

```
g(P,lista(regla(const(f',lista(var(cero),vacio)),lista(const(f,lista(var(cero))),vacio)),
    lista(regla(const(f',lista(const(repite,lista(var(cero),vacio)),vacio)),
    lista(const(f',lista(var(cero),vacio)),vacio)),P))).
```

Utilizaremos la notación $repite^n(x)$ para referirnos al término de la forma $repite(repite(...repite(x)))$ formado por $n$ veces la constructora *repite* y el término $x$.

Dado $P$ PL-programa, entonces $g(P)$ tendrá las siguientes propiedades:

1. Si $x$ verifica que $f(x) \in M(P)$ entonces $f'(repite^n(x)) \in M(g(P)))$, como podemos demostrar por inducción. De este modo, si asumimos este resultado como cierto (luego lo probaremos), tendremos que si existe un $x$ tal que $x \in M(P)$ entonces habrá infinitos $x'$ tales que $f'(x') \in M(g(P))$.

   De este modo, nos queda únicamente por demostrar que si $f(x) \in M(P)$ entonces $f'(repite^n(x) \in M(g(P)))$ para cualquier $n$. Lo haremos por inducción matemática sobre $n$.

   Si $f(x) \in M(P)$ entonces, por la cláusula $R_1$, tenemos que $f'(x) \in M(g(P))$, por lo que se verifica el caso base.

   Para $n+1$, asumiendo que se cumple en $n$, como nuestro $g(P)$ incluye la cláusula $R_2$ tenemos que al cumplirse que $f'(repite^n(x)) \in M(g(P))$ entonces $f'(repite^{n+1}) \in M(g(P))$.

   Hemos llegado a que $f'(repite^n(x)) \in M(g(P))$ para cualquier $n$, por lo que habrá infinitos $x'$ (todos los de las forma $repite^i(x)$) tales que $f'(x') \in M(g(P))$.

2. Si no existe ningún $x$ tal que $f(x) \in M(P)$ veamos entonces que $\nexists x'$ tal que $f'(x') \in M(g(P))$.

   En caso de que hubiera uno, este tan solo podría formar parte de la semántica como resultado de la cláusula $R_2$, ya que como $f(x) \notin M(g(P))$ para todo $x$ entonces no va a poder ser consecuencia de la cláusula $R_1$ nunca.

   Nuestro término, por tanto, tiene que ser de la forma $repite^k(x)$ cumpliéndose que $f'(repite^{k-1}(x)) \in M(g(P))$.

   Podemos realizar este razonamiento de nuevo tantas veces como queramos, hasta llegar a un $x'$ sobre el que no se haya aplicado ninguna vez la constructora *repite* y que cumpliría que $f'(x') \in M(g(P))$.

   Sin embargo, sobre él no puede haberse aplicado la cláusula

   $R_2$: `f'(repite(X)) :- f'(X).`

   dado que las cabezas no coinciden. Como las únicas cláusulas de las que puede ser instancia son $R_1$ y $R_2$, y no proviene de ninguna hemos llegado a un absurdo. Por lo tanto, podemos concluir con que no existe ningún $x'$ tal que $f'(x') \in M(g(P))$.

Acabamos de demostrar que si $f(x) \in M(P)$ entonces habrá infinitos $x'$ tales que $f'(x') \in M(g(P))$. Sin embargo, en el caso de que $f(x) \notin M(P)$ no habrá ninguno que lo cumpla.

De este modo, tenemos que no existe un $x$ tal que $f(x) \in M(P)$ si y solamente si el número de $x'$ tales que $f'(x') \in M(g(P))$ es finito.

Fijándonos en la definición de estos dos conjuntos vemos que hemos llegado a que $P \in E_f \Leftrightarrow g(P) \in FIN_{f'}$, con lo que $g$ es una PL-reducción de $E_f$ sobre $FIN_{f'}$. □

Un resultado inmediato de que $E_f$ sea PL-reducible a $FIN_{f'}$ es que, dado que conocemos que el primer conjunto es PL-reducible, entonces el segundo también lo será.

**Corolario 6.2.3.** *$FIN_{f'}$ no es PL-recursivo.*

*Demostración.* Consecuencia del corolario 6.2.1. □

Nos gustaría añadir algunas reflexiones acerca de la demostración que acabamos de realizar antes de acabar esta sección.

Más allá de proporcionarnos un nuevo ejemplo de conjunto no PL-recursivo, queremos señalar como en este caso se trata de un conjunto bastante más complejo de los que hemos tratando el la anterior sección. El uso de PL-reducciones nos ha permitido centrarnos en estudiar únicamente la relación entre $FIN_f$ y $E_f$, sin tener que trabajar al mismo tiempo con sus funciones características. No hemos tenido que hacer nada más que demostrar la existencia de la función PL-computable total $g$ para ver la relación existente entre ambos y poder utilizarla para demostrar que $FIN_f$ no es PL-recursivo.

### 6.2.2. Uso de la PL-reducibilidad para encontrar conjuntos no PL-recursivamente enumerables

Hemos visto en la anterior sección que somos capaces de encontrar fácilmente ejemplos conjuntos no PL-recursivos haciendo uso de la noción de la PL-reduciblidad. Pero esto no va a limitarse únicamente a la PL-recursividad, sino que también vamos a poder hacerlo con conjuntos PL-recursivamente enumerables.

Antes reflexionamos acerca de que para demostrar que un conjunto no es PL-recursivamente enumerable en ocasiones tratamos de demostrar que no es PL-recursivo, y así obtener que o bien él o su complementario no son PL-recursivamente enumerables. Pues bien, el uso de PL-reducciones va a permitirnos trabajar de modo directo para probar que un conjunto no es PL-recursivamente enumerable. Vamos a desarrollar un teorema similar al que vimos para la PL-recursividad de conjuntos, ahora estudiando su PL-recursiva enumerabilidad.

Para demostrarlo no vamos a tener que hacer muchos cambios respecto a la demostración desarrollada para el teorema 6.2.1, nos basta con sustituir la función característica por el predicado *en*.

Pasamos a enunciar y demostrar el resultado, que luego aplicaremos de cara a estudiar $IG_f$.

**Teorema 6.2.2.** *Si $A \leq_{PL} B$ y $B$ es PL-recursivamente enumerable, entonces $A$ es PL-recursivamente enumerable.*

*Demostración.* Tenemos que $B$ es PL-recursivamente enumerable, por lo que habrá un PL-programa $P_B$ que compute el predicado $enB$. Por otro lado, sea $P_g$ el PL-programa que computa la PL-reducción entre ellos. Vamos a tratar de formar a partir de ellos un PL-programa $P$ que compute el predicado $enA$.

Tomamos $P_A = P_B \cup P_g \cup \{R\}$ siendo $R$ la cláusula:

```
R: enA(X1,...,Xn) :-
      g(X1,..., Xn,Y1,...,Ym),
      enB(Y1,...,Ym).
```

Veamos que el predicado $enA$ efectivamente se comporta como esperamos.

Se verifica que $g(\overline{x}) = \overline{y} \Leftrightarrow g_P(\overline{x}, \overline{y}) \in M(P_A)$. Estamos realizando un abuso de notación refiriéndonos por $g_P$ al predicado $g$ de $P_A$.

Por otro lado, tenemos que $\overline{y} \in B \Leftrightarrow enB(\overline{y}) \in M(P_A)$.

Juntamos ahora estos resultados. Como $g$ es una PL-reducción tenemos que $\overline{x} \in A \Leftrightarrow g(\overline{x}) \in B$. Ahora bien, esto ocurrirá si y solo si $g_P(\overline{x}, \overline{y}), enB(\overline{y}) \in M(P_A)$. Por nuestra cláusula $R$, al ser la única con el predicado $enA$ en su cabeza, tenemos que $f_P(\overline{x}, \overline{y}), enB(\overline{y}) \in M(P_A) \Leftrightarrow enA(\overline{x}) \in M(P_A)$.

Esto nos lleva a que $\overline{x} \in A \Leftrightarrow enA(\overline{x}) \in M(P_A)$, por lo que $P$ computa el predicado $enA$ que necesitamos. De este modo, $A$ será PL-recursivamente enumerable.

□

Al igual que en el caso de los conjuntos PL-recursivos, nos va a interesar más encontrar ejemplos de conjuntos que no sean PL-recursivamente enumerables que encontrar los que lo sean. Por este motivo, desarrollamos ahora el contrarrecíproco de este teorema, que va a ser el que muchas veces vamos a aplicar.

**Corolario 6.2.4.** *Si $A \leq_{PL} B$ y $A$ no es PL-recursivamente enumerable, entonces $B$ no es PL-recursivamente enumerable.*

*Demostración.* Se trata del contrarrecíproco del teorema 6.2.2 □

Podremos utilizar este resultado para probar que un conjunto $B$ no es PL-recursivamente enumerable desarrollando PL-reducciones sobre conjuntos de los que hayamos probado que no son PL-recursivamente enumerables.

A lo largo de este trabajo hemos visto algunos ejemplos de conjuntos no PL-recursivamente enumerables como $\overline{H_f}$. En el resultado que vamos a desarrollar ahora vamos a ver cómo estos ejemplos nos van a permitir encontrar otros conjuntos no PL-recursivamente enumerables a través de PL-reducciones.

**Proposición 6.2.3.** *Se puede construir un PL-reducción entre el conjunto $\overline{H_f}$ a los conjuntos $IG_{f'}$ y $\overline{IG_{f'}}$. Es decir, se cumple que $\overline{H_f} \leq_{PL} IG_{f'}$ y $\overline{H_f} \leq_{PL} \overline{IG_{f'}}$.*

*Demostración.* Comenzamos estudiando el caso de $IG_{f'}$.

Veamos que $H_f$ es reducible a $\overline{IG_{f'}}$, lo que implicaría por el lema 6.2.1 que $\overline{H_f}$ lo es a $IG_{f'}$.

Tomamos para ello la función $g$ que asigna a cada programa $P$ y término $x \in \mathcal{T}_C$ el par $(P_x, P_{vacio})$, siendo $P_x$ el programa original al que se le ha añadido la cláusula $R$ siguiente:

```
R: f'(Y) :- f(x).
```

De este modo se verifica que $f(x) \in M(P) \Leftrightarrow \exists y : f'(y) \in M(P_x)$, ya que $R$ es la única cláusula con el predicado $f'$ en su cabeza.

La función $g$ va a ser PL-recursivamente enumerable y un PL-programa que la computa será el siguiente:

```
g(P,X,
    lista(regla(const(f',lista(var(cero),vacia)),lista(const(f,lista(X),vacia)),P)),
    p_vacio).
```

Veamos ahora que la función $g$ va a reducir $H_f$ a $\overline{IG_{f'}}$.

Tenemos que $(P, x) \in H_f \Leftrightarrow f(x) \in M(P)$. Ahora bien, esto sucede si y solo si $f'(x) \in M(P_x)$. Como $R$ es la única cláusula con $f'$ en su cabeza podemos ir un paso más allá y afirmar que existe un $x'$ tal que $f'(x') \in M(P_x) \Leftrightarrow f(x) \in M(P)$.

Ahora bien, como $M(P_{vacio}) = \emptyset$ tendremos que entonces que $P_x$ y $P_{vacio}$ tendrán la misma semántica respecto de $f'$ si y solo si $f(x) \notin M(P)$. Es decir, $x \in M(P)$ si y solo si $(P_x, P_{vacio}) = g(P) \notin IG_{f'}$.

De esta manera hemos probado que $H_f \leq_{PL} \overline{IG_{f'}}$, que como hemos visto implica que $\overline{H_f} \leq_{PL} IG_{f'}$.

Pasamos ahora al caso de $\overline{IG_{f'}}$.

Seguiremos un razonamiento muy similar, buscando probar su negación que es que $H_f$ es PL-reducible a $IG_{f'}$. La única diferencia estará en que el PL-programa que usaremos para compararlo con

$P_x$ se tratará del que acepta todo que denotaremos por $P_{total}$ (es decir, aquel que para todo $z$ se tiene $f'(z) \in M(P_{total})$ en vez del programa vacío).

Que este programa existe es muy sencillo de comprobar, nos basta con construirlo como vamos a hacer ahora. Tomamos $P_{total}$ el PL-programa formado únicamente por la cláusula:

```
R: f'(X).
```

De este modo, la función $g$ que buscamos será la que a cada programa $P$ y término $x \in \mathcal{T}_C$ le asocie $(P_x, P_{total})$ siendo $P_x$ el programa original al que se le ha añadido la cláusula:

```
R: f'(Y) :- f(x).
```

al igual que lo definimos en el caso anterior.

Para ver que esta función $g$ es PL-computable nos basta con construir un programa similar al que hicimos en el caso anterior, sustituyendo $P_{vacio}$ por $P_{total}$.

```
g(P,X,
    lista(regla(const(f',lista(var(cero),vacia)),lista(const(f,lista(X),vacia)),P)),
    ,p_total).
```

Probemos ahora que $g$ va a reducir $H_f$ a $IG_{f'}$.

Al igual que en el caso anterior, tenemos que $(P, x) \in H_f \Leftrightarrow f(x) \in M(P)$. Ahora bien, como $R$ es la única cláusula de $P_x$ con $f'$ en su cabeza tenemos que $f(x) \in M(P)$ si y solo si para todo $x'$ se tiene que $f'(x') \in M(P_x)$.

De este modo, queda claro que las semánticas de $P_x$ y $P_{total}$ coincidirán respecto de $f'$ si y solo si $f(x) \in M(P)$. Con esto concluimos que $(P, x) \in H_f \Leftrightarrow (P_x, P_{total}) = g(P) \in IG_{f'}$.

Por lo tanto, hemos reducido $H_f$ sobre $IG_{f'}$, lo que implica que $\overline{H_f} \leq_{PL} \overline{IG_{f'}}$. □

Ahora que hemos visto que podemos establecer una PL-reducción entre $\overline{H_f}$ y estos conjuntos ya es muy sencillo concluir que ninguno de ellos es PL-recursivamente enumerable.

**Corolario 6.2.5.** *Los conjuntos $IG_{f'}$ y $\overline{IG_{f'}}$ no son PL-recursivamente enumerables.*

*Demostración.* Consecuencia del corolario 6.2.4. □

Nos gustaría resaltar cómo en este resultado hemos ido un paso más allá de lo que nos demostraba el hecho de que $IG_{f'}$ no fuera PL-recursivo. Habíamos llegado a que $IG_{f'}$ o $\overline{IG_{f'}}$ no iban a ser PL-recursivamente enumerables. Pues bien, acabamos de demostrar que en este caso no van a serlo ninguno de los dos.

Damos con este ejemplo esta sección por concluida. Ya tenemos numerosos ejemplos de conjuntos no PL-recursivos y no PL-recursivamente enumerables y hemos desarrollado técnicas como las PL-reducciones para encontrarlos.

En el siguiente capítulo vamos a dejar esta búsqueda de ejemplos de lado y centrarnos en estudiar teoremas fundamentales de la Teoría de la Computabilidad clásica y su traducción a nuestra perspectiva de PL-programas. Va a ser interesante comprobar cómo algunas de las demostraciones van a simplificarse enormemente al usar programas lógicos, gracias a que estos nos permiten modificar directamente otros PL-programas añadiendo cláusulas, como ya hemos hecho en alguno de los ejemplos de esta sección.

# Capítulo 7

# Teoremas fundamentales

Vamos a probar en este capítulo una serie de teoremas fundamentales en la Teoría de la Computabilidad sobre nuestras funciones PL-computables. Cada sección del capítulo se va a corresponder con uno de los teoremas, que enunciaremos, demostraremos y de los que finalmente estudiaremos algunas de sus aplicaciones.

Nos gustaría destacar el hecho de que las ideas que vamos a utilizar para demostrarlos ya han aparecido anteriormente a lo largo de este trabajo, no vamos a manejar técnicas más allá de lo que ya conocemos. Sin embargo, seremos capaces de llegar a los principales teoremas de la Teoría de la Computabilidad de una manera sencilla basándonos en ellas.

Aunque se destacará en el desarrollo de los teoremas, nos gustaría señalar que muchas de las demostraciones que pasamos a realizar serán más sencillas que las realizadas a lo largo de la teoría clásica. Esto podremos observarlo especialmente en el caso del Teorema s-m-n.

En otros casos, como el Teorema de Rice o de Recursión, no va a existir una diferencia tan grande en la dificultad de las demostraciones, sino que realmente lo que vamos a hacer es adaptar la demostración clásica al ámbito de los PL-programas.

## 7.1. Teorema de Rice

Dedicaremos esta sección a demostrar un importante resultado sobre computabilidad conocido como Teorema de Rice. El nombre de este teorema se debe al matemático Henry Gordon Rice, que demostró el resultado en su disertación doctoral en el año 1951.

Este teorema se refiere a la PL-recursividad de conjuntos y nos va a permitir establecer una sencilla condición suficiente para que un conjunto no sea PL-recursivo. Hasta ahora hemos tenido que demostrar la no PL-recursividad de cada conjunto uno a uno, pero el teorema de Rice nos va a permitir establecer un sencillo criterio que nos ayude tanto a generar conjuntos no PL-recursivos como a probar que efectivamente lo son.

Hemos tomado como referencia (Sipser, 2012), aunque hemos adaptado tanto el enunciado como la demostración para el ámbito de los PL-programas.

Comenzamos fijando la signatura de constructoras $C$ que definimos en 5.1, sea $\mathcal{T}_C$ los términos básicos que se pueden construir con ella. Tomamos además el conjunto $Prog$ formado por todos los términos de $\mathcal{T}_C$ que representan PL-programas. Podemos comprobar fácilmente mediante el PL-programa $V$ definido en el anexo C que $Prog$ es PL-recursivamente enumerable.

Antes de pasar a nuestro teorema vamos a definir algunas nociones que van a ser fundamentales en él.

En primer lugar, nos gustaría señalar que todo conjunto $A \subseteq \mathcal{T}_C^n$ puede entenderse como una relación entre tuplas de $\mathcal{T}_C^n$. Nosotros estudiaremos ahora conjuntos que estén formados por los términos de $\mathcal{T}_C^n$ que verifiquen una cierta propiedad, es decir, los conjuntos de la forma $A = \{x \in \mathcal{T}_C^n | p(x)\}$ siendo $p$ dicha propiedad. Diremos que en este caso $A$ está definido por la propiedad $p$.

Vamos a establecer ahora una serie de definiciones.

**Definición 7.1.1.** *Sea $S$ un conjunto formado por términos que representan programas, es decir, $S \subseteq Prog$. Decimos que $S$ está definido por una propiedad semántica $p$ sobre un predicado $f$ si la propiedad que lo define $p$ verifica:*

$$(\forall x \in \tau_C^n, f(x) \in M(P_1) \Leftrightarrow f(x) \in M(P_2)) \Leftrightarrow (p(P_1) \Leftrightarrow p(P_2)) \Leftrightarrow (P_1 \in S \Leftrightarrow P_2 \in S)) \tag{7.1}$$

*Decimos que $S$ está definido por una propiedad trivial $p$ si la verifican todos los programas o ninguno, es decir, si $S = Prog$ o $S = \emptyset$.*

Es decir, una propiedad es semántica sobre un predicado si el hecho de que un programa la cumpla o no depende exclusivamente de la semántica de dicho predicado. Por ejemplo, la propiedad de tener 5 cláusulas (es decir, $S = \{P \in \mathcal{T}_C | P$ es PL-programa y tiene 5 clásusulas$\}$) no es semántica, mientras que la propiedad de tener modelo mínimo vacío respecto al predicado (es decir, $S = \{P \in \mathcal{T}_C | P$ es PL-programa $\wedge$ para todo $x \in \mathcal{T}_C, f(x) \notin M(P)\}$) sí que es semántica.

Con esta terminología, el Teorema de Rice nos dice que si un conjunto definido por una propiedad semántica sobre un predicado es decidible (PL-recursivo) entonces la propiedad es trivial.

El teorema constituye por tanto un resultado muy genérico y profundo sobre los límites de lo computable en el ámbito del análisis de programas.

**Teorema 7.1.1** (Teorema de Rice)**.** *Sea $S \subseteq Prog$ conjunto de PL-programas definido por una propiedad semántica $p$ sobre un predicado de los PL-programas. Entonces se tiene que si $S$ es PL-recursivo entonces $p$ es una propiedad trivial.*

*Demostración.* Vamos a demostrarlo haciendo uso de las PL-reducciones que definimos en las sección anterior. Aunque este resultado también podría alcanzarse por reducción al absurdo, usar PL-reducciones nos facilita la tarea.

Sea $S$ conjunto definido por una propiedad $p$ no trivial, construiremos una PL-reducción de $S$ sobre el conjunto $H_g$, que definimos en 5.5.1 y probamos que no era PL-recursivo. Si demostramos que $H_g \leq_{PL} S$ entonces, por el corolario 6.2.1, tendremos que $S$ no será PL-recursivo. Antes de continuar con nuestra demostración, recordamos la definición de nuestro conjunto $H_g$,

$H_g = \{(P, x) | P$ es PL-programa y $g(x) \in M(P)\}$

En primer lugar, denotamos por $P_{vacio}$ al PL-programa que no está formado por ninguna cláusula, evidentemente cumplirá $M(P_{vacio}) = \emptyset$.

Podemos suponer sin pérdida de generalidad que $P_{vacio}$ no verifica $p$, en caso contrario nos bastaría con proceder sobre $\overline{S}$, que también sería también PL-recursivo. Como $S$ es no trivial existirá un PL-programa $P_S$ tal que $P_S$ verifica $p$ sobre su predicado *obj*.

Buscamos encontrar una función PL-computable *convierte* tal que para todo término $x \in \mathcal{T}_C$ se cumpla que $(P, x) \in H_f \Leftrightarrow convierte(P, x) \in S$. Nos basaremos para ello en la capacidad de la función $carS$ de distinguir entre los programas que forman parte de $S$ y los que no, permitiéndonos así distinguir entre $P_{vacio}$ y $P_S$.

Tomamos el PL-programa $P_H = P_S \cup \{R\}$ siendo $R$ la cláusula:

```
R: convierte(P,X,P_x):-
      une(p_x, lista(regla(const(f,lista(var(cero),vacio)),lista(const(g,lista(X,vacio)),
         lista(const(obj,lista(var(cero),vacio)),vacio))),P), p_S).
```

Aunque en un principio las cláusulas puedan parecer complicadas, vamos a ver como realmente no lo son tanto, sino que la aparente dificultad que presentan se debe más a la sintaxis que hemos tomado para representar PL-programas como términos. Tratemos de comprender el funcionamiento de este programa.

En primer lugar, el predicado *convierte* dado un PL-programa $P$ y un término $x$, devolverá el programa $P_x = P \cup P_S \cup \{R_1\}$ siendo $R_1$ la cláusula:

```
f(Z) :-
      g(x),
      obj(Z).
```

Veamos que esta función *convierte* computada por $P_H$ es una PL-reducción de $H_g$ sobre $S$.

Estudiamos para ello el comportamiento que tendría el PL-programa $P_H$. Para $x \in \mathcal{T}_C$ término cualquiera tenemos que:

1. Si $g(x) \notin M(P)$ entonces para todo $z \in \mathcal{T}_C$ se tiene que $f(z) \notin M(P_x)$, dado que $g(x) \notin M(P_x)$ y $R$ es la única cláusula del programa de cuya cabeza puede ser instancia el predicado $f$.

   De este modo, si $g(x) \notin M(P)$ entonces $P_x$ se comportará de modo similar a $P_{vacio}$ respecto al predicado $f$, con lo que $P_x$ no verificará la propiedad $p$ ya que $P_{vacio}$ no lo hace y $p$ es propiedad semántica.

2. Si $g(x) \in M(P)$ tenemos que para todo $z \in \mathcal{T}_C$, como $R$ es la única cláusula de $P_x$ con $f$ en su cabeza, se cumple que $f(z) \in M(P_x)$ si y solo si $obj(z) \in M(P_x)$.

   Por otro lado, como el predicado *obj* solo aparece en la cabeza de cláusulas de $P_S$ y hemos hecho la unión de manera que no coincidan los predicados de distintos programas tenemos que $obj(z) \in M(P_x) \Leftrightarrow obj(z) \in M(P_S)$.

   De este modo, si $g(x) \in M(P)$ entonces el predicado $f$ de $P_x$ se comportará de un modo similar a *obj* en $P_S$, ya que $f(z) \in M(P_x) \Leftrightarrow obj(z) \in M(P_S)$. Como $P_S$ verificaba $p$ sobre *obj*, tenemos que $P_x$ verificará dicha propiedad sobre $f$.

Acabamos de comprobar que $P_x$ cumplirá $S$ sobre el predicado $f$ si y solo si $g(x) \in M(P)$. Este paso es fundamental en nuestra demostración porque nos permite distinguir si $g(x)$ pertenece o no a $M(P)$ según $P_x$ verifique o no la propiedad $p$. De este modo, tenemos que $(P, x) \in H_g \Leftrightarrow convierte(P, X) = P_x \in S$.

Hemos probado que $H_g \leq_{PL} S$, lo que nos lleva a que como $H_g$ no es PL-recursivo entonces $S$ tampoco lo es.

□

Acabamos de demostrar uno de los resultados clásicos de la Teoría de la Computabilidad sobre nuestros PL-programas, siguiendo para ello un esquema de demostración bastante similar al clásico pero adaptándolo a nuestro contexto de PL-programas. Vemos como más allá de las dificultades en la notación el resultado ha sido bastante rápido de probar y no ha involucrado ninguna técnica más allá de las PL-reducciones que hemos venido haciendo a lo largo de este trabajo.

Una aplicación sencilla e intuitiva de este teorema es encontrar conjuntos que no sean PL-recursivos. Para ello basta con encontrar una propiedad no trivial sobre un predicado de la semántica y el conjunto

formado por los PL-programas que la verifiquen no será PL-recursivo.

De este modo, si tomamos por ejemplo la propiedad $p := |\{x : f(x) \in MP\}| < 10$ tendremos que el conjunto de programas que la verificarían $P$ no será PL-recursivo. Un resultado que en un principio nos podía parecer muy complicado de demostrar (ya que no vemos un conjunto no PL-recursivo sobre el que reducirlo) se ha convertido en inmediato.

Podríamos haber utilizado este resultado para demostrar que algunos de los conjuntos que hemos estudiado como $E_f$ o $FIN_f$ (desarrollados en 6.1.1 y 6.2.2 respectivamente) son no PL-recursivos. Nos habría bastado con encontrar la propiedad que los genera ($\{x : f(x) \in MP\} = \emptyset$ y $\{x : f(x) \in MP\}$ es finito) y comprobar que es no trivial y sobre el predicado $f$ de la semántica. Sin embargo, hemos considerado conveniente desarrollar dichos ejemplos previamente porque nos permitían observar cómo utilizar reducciones de cara a demostrar la no PL-recursividad de conjuntos. En ellos construimos reducciones sobre $H_f$ más sencillas e intuitivas que la que hemos visto en la demostración del Teorema de Rice y que nos han servido como paso previo para este teorema.

Nos gustaría señalar ahora todo el avance que hemos hecho en las demostraciones de la no PL-recursividad de conjuntos.

En un principio, en el caso de $H_f$ nos vimos obligados a utilizar un argumento diagonal para demostrar que no era PL-recursivo. Después pasamos a ser capaces de demostrar que ciertos conjuntos no eran PL-recursivos mediante reducciones, como vimos con $E_f$ o $IG_f$. Fuimos capaces de generalizar parte de estas demostraciones mediante las PL-reducciones, que nos permitían únicamente centrarnos en demostrar que los conjuntos están relacionados.

Ahora, al demostrar el Teorema de Rice hemos ido un paso más allá, desarrollando un resultado que nos permite establecer si un conjunto de PL-programas es PL-recursivo estudiando únicamente cuál es la propiedad que lo define. Hemos pasado de tener que recurrir a argumentos diagonales y demostraciones complejas a poder comprobar si un conjunto es PL-recursivo observando únicamente si la propiedad que lo genera es no trivial y sobre un predicado de la semántica. Esto sin duda representa un gran avance.

## 7.2. Teorema s-m-n

Dedicaremos esta sección a estudiar uno de los teoremas fundamentales de la Teoría clásica de la Computabilidad. Se trata del Teorema s-m-n, también conocido como Teorema del Parámetro, demostrado por el matemático Stephen Cole Kleene en el año 1943.

La importancia de este teorema radica en que nos permite expresar funciones de varias variables mediante otras de un número inferior de variables, fijando para ello alguna de las variables originales. Este resultado es fundamental en la teoría clásica de computabilidad porque se utiliza para realizar reducciones. Sin embargo, nosotros no hemos tenido que recurrir a él para hacerlas, ya que en nuestros ejemplos de reducciones estábamos siguiendo en cierto modo unas ideas similares a las que van a utilizarse en la demostración del Teorema s-m-n.

Antes de pasar a desarrollar el teorema, nos gustaría resaltar cómo la demostración para PL-programas va a ser mucho más sencilla a las desarrolladas para máquinas de Turing o máquinas de registros que puede consultarse en (Cutland, 1980). La demostración va a requerir simplemente construir sencillos PL-programas que computen las funciones indicadas en el teorema.

Que esta demostración vaya a simplificarse tanto se debe a la capacidad de los PL-programas de trabajar con otros PL-programas como términos, pudiendo añadir en ellos nuevas cláusulas o concatenarlos. Hemos utilizado esta capacidad al construir reducciones, generando programas que manipulaban otros. Vamos a ver ahora que también esta va a ser la idea fundamental en nuestra demostración del Teorema s-m-n.

**Teorema 7.2.1** (Teorema s-m-n)**.** *Para todo PL-programa $P$ que compute una función $f$ de aridad $m+n$, y todos los posibles valores de $y_1, ..., y_m \in \mathcal{T}_C$ tenemos que existe un PL-programa $P_k$ que define un predicado $f'$ de aridad $n$ tal que*

$$\forall z_1, \ldots, z_n \in \tau_C,\ f(y_1, \ldots, y_m, z_1, \ldots, z_n) \in M(P) \Leftrightarrow f'(z_1, \ldots, z_n) \in M(P_k) \tag{7.2}$$

*Más aún, la función que denotaremos por $k$ que para cada $P$, $y_1, \ldots, y_m$ nos devuelve su $P_k$ correspondiente, definida por $k(P, y_1, \ldots, y_m) = P_k$, es PL-computable.*

*Es decir, existe un PL-programa $Q$ tal que $k(P, y_1, \ldots, y_m, P_k) \in M(Q)$ si y solamente si $P, y_1, \ldots, y_m$ y $P_k$ verifican la condición anterior.*

*Demostración.* Vamos a proceder de modo constructivo desarrollando el PL-programa $Q$ del que hemos hablado en el enunciado de nuestro teorema.

Antes de hacerlo, de cara a simplificar nuestra notación, vamos a utilizar las siguientes abreviaturas:

1. Nos referiremos por $L_1$ a la lista formada por los términos $Y_1, \ldots, Y_m, z_1, \ldots, z_n$, comportándose las $Y_k$ como constantes y las $z_k$ como variables, que quedaría escrita en nuestra sintaxis PL como $lista(const(Y_1, vacio), lista(const(Y_2, vacio), \ldots, lista(var(suc^{n-1}(cero)), vacio)))$.
2. Por otro lado, a la lista formada por las variables $z_1, \ldots, z_n$ nos referiremos por $L_2$, siendo de la forma $lista(var(cero), lista(var(suc(cero)), \ldots, lista(var(suc^{n-1}(cero)), vacio)))$.

Pasamos ahora a desarrollar el programa $Q$ que computa $k$. Lo definimos como el PL-programa únicamente formado por la cláusula:

`k(P,`$Y_1$`,...,`$Y_m$`,lista(P,lista(regla(const(f',`$L_2$`),lista(const(f,`$L_1$`),vacio)),vacio))).`

La función $k$, por tanto, será la que dado un PL-programa $P$ y los términos $y_1, \ldots, y_m$ nos devuelva el mismo programa $P$ añadiéndole al final la cláusula $R$ dada por

`R: f'(`$Z_1$`,...,`$Z_n$`) :- f(`$y_1$`,...,`$y_m$`,`$Z_1$`,...,`$Z_n$`).`

Veamos que este PL-programa $P_k = P \cup \{R\}$ va a comportarse como esperamos, computando el predicado $f'$ de modo que para todo $z_1, \ldots, z_n \in \mathcal{T}_C$ se tenga que $f(y_1, \ldots, y_m, z_1, \ldots, z_n) \in M(P) \Leftrightarrow f'(z_1, \ldots, z_n) \in M(P_k)$.

Dado que $R$ es la única cláusula de $P_k$ que tiene $f'$ en su cabeza, tenemos que $f'(z_1, \ldots, z_n) \in M(P_k)$ si y solo si $f(y_1, \ldots, y_m, z_1, \ldots, z_n) \in M(P)$.

De este modo, el programa $P_k$ cumple la condición del enunciado, por lo que efectivamente el programa $Q$ computa la función $k$, con lo que $k$ es PL-computable.

□

Vemos que la demostración que acabamos de realizar se parece a algunas de las que hemos venido haciendo. La idea de utilizar programas que añaden cláusulas a los programas originales la hemos utilizado en demostraciones como 6.1.1 o 6.2.2 para estudiar construir reducciones entre conjuntos.

Nos gustaría resaltar cómo en la teoría clásica de Turing-computabilidad se tiene que recurrir muy habitualmente al Teorema s-m-n para hacer reducciones. En nuestro caso no hemos precisado de este teorema, pero sí hemos acabado utilizando las mismas ideas que este usa en su demostración. Aunque no de una manera tan directa como en la teoría clásica, vemos que la relación entre el Teorema s-m-n y las reducciones sigue estando presente.

De este modo, al poder realizar estas construcciones de manera directa podemos decir que, en cierta manera, el Teorema de s-m-n va a tener una menor importancia en el ámbito de los PL-programas que la que puede tener en el caso de las máquinas de Turing o las máquinas de registros. Esto se debe a que, dada la sencillez del PL-programa que computa la función $k$, muchas veces va a ser más

sencillo trabajar directamente sobre las cláusulas de $Q$, añadiendo exactamente las que necesitemos para resolver el problema concreto.

Sin embargo, el Teorema s-m-n va a seguir resultándonos útil para obtener ciertos resultados como el Teorema de Recursión o el del Punto Fijo. Vamos a desarrollar estos resultados en las secciones siguientes, pero antes de pasar a ellos vamos a resaltar un último aspecto acerca del Teorema s-m-n.

**Corolario 7.2.1.** *La función $k$ es inyectiva.*

*Demostración.* Aunque este resultado pueda parecer trivial, vamos a desarrollarlo porque más adelante nos resultará útil.

Tenemos que dado un PL-programa $P$ y los terminos $y_1, ..., y_m$, la función $k$ nos devuelve el mismo programa $P$ añadiéndole al final la cláusula $R$ dada por

R: $f'(Z_1,\ldots,Z_n)$ :- $f(y_1,\ldots,y_m,Z_1,\ldots,Z_n)$.

Veamos que si $k(P, y_1, \ldots, y_m) = k(P', y'_1, \ldots, y'_m)$ todos los argumentos van a coincidir, lo que demostraría que es inyectiva.

Ambos programas $k(P, y_1, \ldots, y_m)$ y $k(P', y'_1, \ldots, y'_m)$ van a estar formados por los programas $P$ y $P'$ concatenados por las cláusulas $R$ y $R'$ respectivamente, siendo estas de la forma mencionada anteriormente. Para que los programas coincidan, deberán coincidir tanto $P$ con $P'$ como las cláusulas $R$ con $R'$. Si $R = R'$ entonces los argumentos $y_1, \ldots, y_m$ tendrán que coincidir uno a uno con $y'_1, \ldots, y'_m$.

Por lo tanto llegamos a que $P = P'$ y $y_k = y'_k$ para $k = 1, \ldots, m$, por lo que $k$ es inyectiva.

□

Comprobaremos ahora que el Teorema s-m-n va a resultar fundamental de cara a demostrar otros resultados como el Teorema de Recursión y el del Punto Fijo que pasamos a desarrollar ahora. Como hemos venido haciendo, vamos a enunciar y demostrar estos teoremas, así como ver algunas de sus aplicaciones.

## 7.3. Teorema de Recursión

Vamos ahora con otro teorema fundamental de la Teoría de la Computabilidad conocido como Teorema de Recursión. Este teorema fue desarrollado por primera vez por Stephen Kleene en el año 1938.

Este resultado nos va a permitir estudiar las funciones de $n + 1$ variables y ver que cada una de ellas simula la ejecución de un PL-programa de $n$ variables. Aunque este resultado puede parecernos de inicio algo completamente abstracto, veremos que va a servirnos para demostrar el Teorema del Punto Fijo y otros resultados interesantes.

Pasamos a enunciar y demostrar el teorema. Nos va a resultar bastante sencillo una vez disponemos del Teorema s-m-n.

**Teorema 7.3.1** (Teorema de Recursión)**.** *Sea $g : \mathcal{T}_C^{n+1} \longrightarrow \mathcal{T}_C$ función parcial PL-computable, entonces existe un PL-programa $E$ tal que en uno de sus predicados $f'$ verificará*

$$\forall x_1, \ldots, x_n \in \tau_C, g(E, x_1, \ldots, x_n) = z \Leftrightarrow f'(x_1, \ldots, x_n, z) \in M(E) \tag{7.3}$$

*Demostración.* En primer lugar, antes de pasar a nuestra demostración, denotaremos por $k$, al igual que en 7.2.1, a la función PL-computable dada por el Teorema s-m-n aplicado sobre aridades 1 y $n$. Tenemos, por tanto, que se verificará $\forall x_1, \ldots, x_n \in \mathcal{T}_C$, $f(u, x_1, \ldots, x_n) \in M(P) \Leftrightarrow f'(x_1, \ldots, x_n) \in M(P_k)$

siendo $P_k = k(P, u)$.

Una vez que tenemos nuestra $k$ definida, tomamos la función $f$ de manera que

$$f(P, x_1, \ldots, x_n) = g(k(P, P), x_1, \ldots, x_n) \tag{7.4}$$

Al ser $f$ sustitución de funciones PL-computables será PL-computable (vimos en 3.2.1 que las funciones PL-computables son cerradas respecto a la sustitución, recursión y minimalización). Por este motivo podemos asegurar que existirá un PL-programa $P_f$ que la compute, de manera que $P_f$ cumplirá $g(k(P, P), x_1, ..., x_n) = z \Leftrightarrow f(P, x_1, ..., x_n, z) \in M(P_f)$ (estamos haciendo un abuso de notación refiriéndonos por $f$ tanto a la función como al predicado del programa $P_f$).

Ahora bien, por el Teorema s-m-n, llegamos a que al aplicar la función $k$ sobre $P_f$ y $P$ se obtiendrá un PL-programa $P_k = k(P_f, P)$ que verificará para todo $x_1, ..., x_n \in \mathcal{T}_C$ que $f(P, x_1, \ldots, x_n) \in M(P_f) \Leftrightarrow f'(x_1, \ldots, x_n) \in M(P_k)$.

Combinando ambas igualdades, llegamos a que

$$g(k(P, P), x_1, \ldots, x_n) = z \Leftrightarrow f(k(P, P), x_1, \ldots, x_n, z) \in M(P_f) \Leftrightarrow f'(x_1, ..., x_n) \in M(P_k), P_k = k(P_f, P) \tag{7.5}$$

$P_f$ es un PL-programa fijo y se corresponde con el que computa la función $f$. En cambio $P$ es variable. Por este motivo, podemos tomar finalmente $P = P_f$ y denotando por $E$ al programa $k(P_f, P_f)$ llegamos al resultado buscado.

Nos basta con sustituir en la ecuación 7,5 los valores que acabamos de fijar, llegando así a que $g(E, x_1, ..., x_n) = z \Leftrightarrow f'(x_1, ..., x_n) \in M(E)$. □

Como podemos ver en la demostración, este teorema es consecuencia directa del Teorema s-m-n. Esto también sucede en el contexto de la Turing-computabilidad clásica y nos muestra un ejemplo de cómo el Teorema s-m-n sigue siendo importante en nuestra teoría.

Centrándonos ya completamente en el Teorema de Recursión, nos gustaría señalar que, aunque en un principio nos pueda parecer algo completamente abstracto y sin mucha utilidad, en lo que nos queda de capítulo vamos a comprobar que no es así. Pero antes de pasar a ver sus aplicaciones vamos a ver cómo podemos ir un paso más allá en lo que enuncia el Teorema de Recursión, llegando a que no solo va a existir un PL-programa que lo cumpla, sino que van a ser infinitos.

Este resultado es muy sencillo de probar, tan solo tenemos que incluir en la demostración que acabamos de realizar un pequeño razonamiento basado en el hecho de que la función $k$ es inyectiva, como demostramos en el corolario 7.2.1.

**Corolario 7.3.1.** *Sea $g : \mathcal{T}_C^{n+1} \longrightarrow \mathcal{T}_C$ parcial PL-computable, entonces existen infinitos PL-programas $E$ tales que*

$$f'(x_1, \ldots, x_n, z) \in M(E) \Leftrightarrow g(E, x_1, \ldots, x_n) = z \tag{7.6}$$

*Demostración.* Observando la demostración anterior vemos que se basa en que la función $f$ definida por $f(P, x_1, \ldots, x_n) = g(k(P, P), x_1, \ldots, x_n)$ es PL-computable, por lo que existirá un PL-programa $P_f$ que la compute.

Esto es sin duda cierto, pero podemos ir un paso más allá y afirmar que no solo va a existir un PL-programa que la compute, sino que va a haber infinitos. Nos basta con observar que podemos añadir a $P_f$ cláusulas que no afecten de ningún modo a nuestro predicado $f$ (por ejemplo, añadiendo cláusulas de la forma $suc^i(cero) : -suc^i(cero)$.). De esta manera, estaríamos construyendo infinitos PL-programas $P'_f$ que computan la función $f$.

La $k$ dada por el Teorema s-m-n es inyectiva. Por este motivo, cada uno de nuestros $P'_f$ genera un $E' = k(P'_f, P'_f)$ distinto. Podemos así concluir que existirán infinitos $E'$ que cumplan $g(E', x_1, ..., x_n) = z \Leftrightarrow f'(x_1, ..., x_n) \in M(E')$.

□

Antes de terminar esta sección vamos a desarrollar un ejemplo que nos va a permitir ver un uso práctico de estos teoremas. Creemos que esto puede ser interesante porque nos permite ver que, aunque estamos desarrollando estos resultados de manera bastante teórica, sí tienen relevancia sobre problemas prácticos que nos plantea el trabajar sobre PL-programas.

**Ejemplo 7.3.1.** *Un problema habitual en la Teoría clásica de la Computabilidad es el de los programas que devuelven como resultado el propio programa. Vamos a tratar en este ejemplo de plasmar esta idea sobre nuestros PL-programas. Aunque pueda parecer sorprendente, la existencia de este tipo de programas se puede demostrar mediante el Teorema de Recursión.*

*Comenzamos definiendo cuáles son los PL-programas que vamos a estudiar. Se trata de aquellos que para cualquier argumento nos devuelvan el propio programa, verificando así* $\forall x \in \mathcal{T}_C \ f(x,y) \in M(P) \Leftrightarrow y = P$. *Se corresponden con la traducción al ámbito de la PL-computabilidad de los programas autorreplicantes, que son los que reproducen su propio código.*

*Aunque diseñar programas que cumplan esta condición es complicado, gracias al corolario que acabamos de probar nos va a resultar sencillo demostrar no solo que van a existir, sino también que van a ser infinitos.*

**Proposición 7.3.1.** *Existen infinitos PL-programas* $E$ *que verifican* $f'(y,z) \in M(E) \Leftrightarrow z = E$ *para todo* $y \in \mathcal{T}_C$ .

*Demostración.* Comenzamos tomando la función $g : \mathcal{T}_C^2 \to \mathcal{T}_C$ definida por $g(x,y) = x$ para todo $x, y \in \mathcal{T}_C$.

Esta función es evidentemente PL-computable, por lo que por el Teorema de Recursión tenemos que existirán infinitos PL-programas $E$ tales que para todo $y$ cumplan $g(E,y) = z \Leftrightarrow f'(y,z) \in M(E)$.

Ahora bien, por definición para todo $y$ se tiene que $g(E,y) = E$, por lo que esta condición nos queda como $f'(y,z) \in M(E) \Leftrightarrow z = E$.

□

*Acabamos de ver que el Teorema de Recursión nos permite asegurar que van a existir PL-programas autorreplicantes. Aunque no forma realmente parte de lo que estudiamos en esta sección, vamos a ver otro resultado interesante acerca de este tipo de PL-programas.*

*Hemos demostrado que existen infinitos PL-programas autorreplicantes aunque no hemos construido ninguno. Esto nos puede llevar a preguntarnos si seríamos capaces de decidir cuándo un PL-programa es autorreplicante.*

*Esta es la cuestión sobre la que ahora vamos a reflexionar brevemente. Tomamos* $AUTO_f$ *el conjunto formado por los PL-programas autorreplicantes en uno de sus predicados* $f$. *De este modo* $AUTO_f = \{P : \forall x \ f(x,y) \in M(P) \Leftrightarrow y = P\}$. *Demostraremos ahora haciendo uso del Teorema de Recursión que este conjunto no va a ser PL-recursivo.*

**Proposición 7.3.2.** *El conjunto* $AUTO_f$ *no es PL-recursivo.*

*Demostración.* Supongamos que fuera PL-recursivo, en tal caso la función característica de $AUTO_f$ sería PL-computable.

De este modo, estamos suponiendo que la función

$$h(P) = \begin{cases} suc(cero), & \text{sii P es un programa autorreplicante en f} \\ cero, & \text{sii P no es un programa autorreplicante en f} \end{cases}$$

es PL-computable, denotamos por $P_h$ al PL-programa que la computa.

Tomamos un PL-programa $P_0$ de manera que no sea autorreplicante (por ejemplo, nos basta tomar $P_0 = vacio$, es decir, el PL-programa vacío).

Ahora, tomamos $g$ la función dada por

$$g(P,x) = \begin{cases} P_0, & \text{si} \quad h(P) = suc(cero) \\ P, & \text{si} \quad h(P) = cero \end{cases}$$

Esta nueva función también es PL-computable, como nos es muy sencillo de comprobar observando que la computa el PL-programa $P_g = P_h \cup \{R_1, R_2\}$, siendo estas las cláusulas siguientes:

```
R1: g(P,X,p0) :- h(P,suc(cero)).
R2: g(P,X,P) :- h(P,cero).
```

Aplicando el Teorema de Recursión sobre la función $g$ llegamos a que existe un PL-programa $E$ tal que $f(x,z) \in M(E) \Leftrightarrow g(E,x) = z$. Comprobemos que esto nos lleva a un absurdo.

1. Si $E$ es autorreplicante en un predicado $f$ entonces $f(x,z) \in M(E) \Leftrightarrow z = E$.
   Pero como $g(E,x) = P_0$, tendríamos por otro lado que $f(x,P_0) \in M(E)$.
   Esto es absurdo, porque implicaría que $E = P_0$, lo que es imposible al ser uno autorreplicante y el otro no.
2. Si $E$ no es autorreplicante en $f$ entonces $g(E,x) = E$.
   Tenemos que $g(E,x) = E \Leftrightarrow f(x,E) \in M(E)$. De esta manera hemos llegado a un absurdo porque para todo $x \in \mathcal{T}_C$, $f(x,y) \in M(E) \Leftrightarrow y = E$, con lo que $E$ sería autorreplicante.

Como ambas posibilidades nos llevan a una contradicción, $h$ no puede ser PL-computable.

□

*Observamos cómo haciendo uso del Teorema de Recursión hemos sido capaces de probar que $AUTO_f$ no va a ser PL-recursivo de manera directa, sin hacer uso de ninguna reducción.*

*Además, nos gustaría señalar que el conjunto $AUTO_f$ a diferencia de la mayoría de los conjuntos no PL-recursivos que hemos estudiado, no se encuentra dentro de los supuestos del teorema de Rice. La condición de ser autorreplicante en $f$, como podemos observar fácilmente, no es una condición sobre la semántica de los PL-programas. Esto hace que el Teorema de Rice sea inútil en este caso.*

*De este modo, el Teorema de Recursión nos ha permitido demostrar fácilmente que existen más conjuntos no PL-recursivos que los dados por el Teorema de Rice.*

Damos por finalizado este ejemplo, que nos ha permitido observar aplicaciones bastante sorprendentes del Teorema de Recursión. Sin embargo, este teorema nos va a llevar a resultados más importantes. Comprobaremos en la siguiente sección como mediante el Teorema de Recursión podemos llegar a uno de los teoremas fundamentales de la Teoría de la Computabilidad: el Teorema del Punto Fijo.

En la demostración de este teorema combinaremos el Teorema de Recursión con el Teorema s-m-n, con el objetivo de probar la existencia de puntos fijos en las funciones PL-computables.

## 7.4. Teorema del Punto Fijo

Acabamos este capítulo desarrollando otro de los teoremas fundamentales de la Teoría de la Computabilidad: el Teorema del Punto Fijo. Este nos va a permitir estudiar los puntos fijos respecto a un predicado que van a presentar las funciones PL-computables al aplicarse sobre términos que representen PL-programas.

El Teorema del Punto Fijo fue demostrado por Stephen Kleene, al igual que el Teorema de Recursión. Esto ha llevado a que el Teorema del Punto Fijo también se conozca como Segundo Teorema de Recursión.

Recordamos que el concepto de punto fijo de una función $f$ se refiere a un valor $x$ tal que $f(x) = x$. En el caso de los términos que representan PL-programas no vamos a pedir tanto en este teorema, no exigiremos que $P$ y $f(P)$ coincidan, sino que pediremos que tengan la misma semántica para uno de sus predicados. Es decir, que para uno de sus predicados $g$ se tenga que $g'(x) \in M(P) \Leftrightarrow g(x) \in M(f(P))$ para todo $x$.

El Teorema del Punto Fijo nos permite estudiar funciones PL-computables definidas entre términos que representan PL-programas, es decir, funciones PL-computables $f$ definidas en $f : Prog \longrightarrow Prog$ y demostrar que siempre va a existir un PL-programa $E$ que se comporte como un punto fijo. Por $Prog$ nos estamos refiriendo al conjunto formado por todos los términos de $\mathcal{T}_C$ que representan PL-programas, como hicimos anteriormente.

Vamos a comenzar la sección enunciando y probando una versión más sencilla del teorema, de cara a dejar clara la estructura que seguirá nuestra demostración.

**Teorema 7.4.1.** *Sea $f : Prog \longrightarrow Prog$ función PL-computable y sea $g$ uno de sus predicados, entonces existe un PL-programa $E$ tal que para todo $x$ se cumple que $g'(x) \in M(E) \Leftrightarrow g(x) \in M(f(E))$.*

*Demostración.* Vamos a demostrarlo mediante el Teorema de Recursión.

Consideramos la función parcial $h : \mathcal{T}_C^2 \longrightarrow \mathcal{T}_C$ definida únicamente para los términos de $Prog \times \mathcal{T}_C$ como $h(P, x) = z \Leftrightarrow g(x, z) \in M(f(P))$. Demostraremos ahora que esta función es PL-computable gracias a la existencia de un PL-programa universal.

Nos basta con tomar el PL-programa $Q = U \cup P_f \cup \{R\}$ siendo $U$ nuestro PL-programa universal, $P_f$ el PL-programa que computa $f$ (al ser PL-computable sabemos que existe) y $R$ la regla:

```
R: h(P,X,Z) :-
     f(P,P1),
     resuelve_uno(const(g,lista(const(X,vacio),lista(const(Z,vacio),vacio))),P1).
```

Para comprobar que el programa efectivamente computa la función $h$ nos basta con seguir un razonamiento similar al que hemos hecho en otras demostraciones.

Tenemos que como $R$ es la única cláusula de $Q$ con el predicado $h$ en su cabeza entonces se verifica que $h(P, x, z) \in M(Q)$ si y solo si existe un término $P_1$ tal que $f(P, P_1) \in M(Q)$ y $resuelve\_uno(const($ $g, lista(const(x, vacio), lista(const(z, vacio), vacio))), P_1) \in M(Q)$.

Como $Q$ computa la función $f$ entonces tenemos que $P_1 = f(P)$. Además, como $Q$ incluye nuestro PL-programa universal $U$, por lo que demostramos en el teorema 4.2.1 se cumplirá que $resuelve\_uno(const(g, lista(const(x, zero), lista(const(z, vacio), vacio))), P_1) \in M(Q)$ si y solo si $g(x, z) \in M(f(P))$. Por lo tanto, $Q$ computa la función $h$.

Acabamos de demostrar que $h$ es PL-computable, por lo que podemos aplicar sobre ella el Teorema de Recursión.

Llegamos así a que existirá un PL-programa $E$ tal que para todo $x$ se cumplirá $g'(x, z) \in M(E) \Leftrightarrow h(E, x) = z$.

Por otro lado, por definición de $h$ tenemos que $h(P, x) = z \Leftrightarrow g(x, z) \in M(f(P))$.

Si unimos ambas expresiones llegamos a que para todo $x$ se cumple $g'(x, z) \in M(E) \Leftrightarrow g(x, z) \in$

$M(f(E))$. Por lo tanto, $E$ es el PL-programa que estamos buscando. □

Esta primera versión del teorema también se conoce como Teorema del Punto Fijo de Rogers y podríamos haberla demostrado sin hacer uso del Teorema de Recursión sino tan solo del Teorema s-m-n. Sin embargo, hemos preferido desarrollar directamente la versión de la demostración que hace uso del Teorema de de Recursión porque vamos a poder generalizarla muy fácilmente, como haremos en lo que queda de sección.

Queremos resaltar que a pesar de que el teorema se denomine Teorema del Punto Fijo, la $E$ que estamos obteniendo no se corresponde realmente con un punto fijo de la función. No hemos demostrado que $E = f(E)$ ni que sus semánticas sean iguales, sino que hemos probado que se van a comportar igual respecto a uno de sus predicados. Es decir, que $g(x) \in M(E) \Leftrightarrow g'(x) \in M(f(E))$. De este modo, no estamos trabajando con puntos fijos, sino con pseudopuntos fijos, a pesar de que el nombre del teorema nos pueda llevar a pensar lo contrario.

Podemos generalizar este teorema fácilmente para predicados de cualquier aridad, dado que el Teorema de Recursión está definido en términos de funciones de cualquier número de argumentos. Esta versión generalizada es la que usualmente se conoce como Teorema del Punto Fijo.

**Teorema 7.4.2** (Teorema del Punto Fijo)**.** *Sea $f$ : Prog $\longrightarrow$ Prog PL-computable y $g$ un de sus predicado cualquiera, entonces existe un PL-programa $E$ tal que para todo $x_1, \ldots x_n$ se cumple que $g'(x_1, \ldots, x_n) \in M(E) \Leftrightarrow g(x_1, \ldots, x_n) \in M(f(E))$.*

*Demostración.* Vamos a comprobar que apenas vamos a tener que modificar nuestra demostración.

Al igual que en el caso anterior consideramos la función parcial $h : \mathcal{T}_C^2 \longrightarrow \mathcal{T}_C$ definida únicamente para los términos de $Prog \times \mathcal{T}_C$ por $h(P, x_1, \ldots, x_n) = z \Leftrightarrow g(x_1, \ldots, x_n, z) \in M(f(P))$, de la que probaremos que es PL-computable de modo similar. Tomamos el PL-programa $Q = U \cup P_f \cup \{R\}$ siendo $R$ la cláusula:

```
R: h(P,X1,...,Xn,Z) :-
     f(P,P1),
     resuelve_uno(const(g,lista(const(X1,vacio),lista(...,lista(const(Z,vacio),vacio)))),P1).
```

Para comprobar que este programa efectivamente computa la función $h$ nos basta con seguir un razonamiento similar al realizado para el caso de aridad 1.

Como $h$ es PL-computable, al aplicar sobre ella el Teorema de Recursión llegamos a que existirá un PL-programa $E$ tal que para todo $x_1, \ldots, x_n \in \mathcal{T}_C$ cumplirá $g'(x_1, \ldots, x_n, z) \in M(E) \Leftrightarrow h(E, x_1, \ldots, x_n) = z$.

Por otro lado, por definición de $h$ tenemos que $h(P, x_1, \ldots, x_n) = z \Leftrightarrow g(x_1, \ldots, x_n, z) \in M(f(P))$.

Unimos igualmente ambas expresiones, obteniendo que para todo $x_1, \ldots, x_n$ se verificará que $g'(x_1, \ldots, x_n, z) \in M(E) \Leftrightarrow g(x_1, \ldots, x_n, z) \in M(f(P))$, con lo que queda demostrado que $E$ es un pseudopunto fijo. □

Acabamos de demostrar una versión general de nuestro teorema, pero aun vamos a ser capaces de ir más allá. Esta nueva generalización no es tan importante como la que acabamos de hacer, pero de todos modos consideramos necesario incluirla porque refleja como los corolarios que hemos demostrado del Teorema s-m-n y de Recursión los podemos aplicar en este nuevo teorema.

En estas demostraciones estamos usando el Teorema de Recursión para encontrar un PL-programa $E$ que verifique $g'(x_1, \ldots, x_n, z) \in M(E) \Leftrightarrow h(E, x_1, \ldots, x_n) = z$. Sin embargo, pudimos ver en la sección dedicada a este teorema que no solo va a ser un $E$ el que lo verifique, sino que van a existir infinitos.

Aplicando esto en la demostración que ya tenemos vamos a llegar a que también van a ser infinitos los puntos fijos que encontremos.

**Corolario 7.4.1.** *Sea $f : Prog \longrightarrow Prog$ PL-computable y $g$ un predicado cualquiera, entonces existen infinitos PL-programas $E$ tales que para todo $x_1, \ldots x_n \in \mathcal{T}_C$ se cumple que $g'(x_1, \ldots, x_n) \in M(E) \Leftrightarrow g(x_1, \ldots, x_n) \in M(f(E))$.*

*Demostración.* Nos basta con mantener prácticamente la misma demostración, modificándola únicamente en el momento en que aplicamos el Teorema de Recursión. Ahora usaremos la versión desarrollada en el corolario 7.3.1 que nos permitía afirmar que eran infinitos los PL-programas que lo cumplían.

Mantenemos así la demostración hasta el punto en que decimos que por ser $h$ PL-computable entonces podemos aplicar sobre ella el Teorema de Recursión. En este caso, vamos a aplicar el corolario obteniendo gracias a él que existirán infinitos PL-programas $E$ tales que para todo $x_1, \ldots, x_n \in \mathcal{T}_C$ cumplirán $g'(x_1, \ldots, x_n, z) \in M(E) \Leftrightarrow h(E, x_1, \ldots, x_n) = z$.

Al igual que en los casos anteriores seguimos teniendo que $h(P, x_1, \ldots, x_n) = z \Leftrightarrow g(x_1, \ldots, x_n, z) \in M(f(P))$.

Al unir ambas expresiones obtenemos que para cada uno de nuestros infinitos $E$ se tiene que para todos $x_1, \ldots, x_n \in \mathcal{T}_C$ se cumplirá que $g'(x_1, \ldots, x_n, z) \in M(E) \Leftrightarrow g(x_1, \ldots, x_n, z) \in M(f(P))$. De esta manera, cada uno de nuestros $E$ será un pseudopunto fijo, con lo que hemos encontrado infinitos. □

Hemos logrado en nuestros corolarios generalizar nuestro teorema, aprovechando todo lo que nos ofrecía el Teorema de Recursión en su forma más general. De este modo, hemos pasado de un teorema tan sólo válido para predicados de un único argumento a otro que se puede aplicar sobre cualquier número de argumentos, además de garantizarnos que van a ser infinitos los pseudopuntos fijos que existen.

Veamos ahora un ejemplo de aplicación del teorema.

Hemos demostrado el Teorema de Rice. La demostración que hicimos en 7.1 estaba basada em construir una PL-reducción entre el conjunto de PL-programas que cumplían la propiedad y $H_f$. La demostración no llegaba a ser complicada, pero sí requería bastante trabajo de cara a probar la PL-computabilidad de la reducción.

Pues bien, vamos a desarrollar ahora una nueva demostración de este teorema como aplicación del Teorema del Punto Fijo. Aunque en un principio puede parecer que estos dos teoremas no están relacionados, vamos comprobar que el de Teorema Punto Fijo nos va a permitir llegar al Teorema de Rice de una manera elegante y menos laboriosa.

**Teorema 7.4.3** (Teorema de Rice)**.** *Sea $S \subseteq Prog$ conjunto de PL-programas definido por una propiedad semántica $p$ sobre un predicado de los PL-programas. Entonces se tiene que si $S$ es PL-recursivo entonces $p$ es una propiedad trivial.*

*Demostración.* Demostraremos el resultado por reducción al absurdo. Suponemos, por tanto, que $p$ es no trivial y que $S$ es PL-recursivo.

Tomamos $P_a$ PL-programa verificando $p$ y $P_b$ que no la verifica, siempre vamos a poder hacerlo ya que $p$ es no trivial. De este modo, $P_a \in S$ y $P_b \notin S$. Si $S$ fuera PL-recursivo, podríamos construir la función $h$ definida por

$$h(P) = \begin{cases} P_a & \text{sii} \quad P \notin S \\ P_b, & \text{sii} \quad P \in S \end{cases}$$

que sería PL-recursiva. Un PL-programa que la computa sería $P = P_S \cup \{R_1, R_2\}$ siendo $P_S$ el programa que computa la función característica de $S$ y $R_1$ y $R_2$ las cláusulas:

```
R1: h(P,pa) :- carS(P,cero).
R2: h(P,pb) :- carS(P,suc(cero)).
```

Nos referimos por $p_a$ al programa $P_a$ y por $p_b$ a $P_b$, hemos tomado esta notación en nuestras cláusulas ya que las mayúsculas se refieren a variables.

Tenemos que $P$ computa la función $h$ como es muy sencillo comprobar. Además, para cualquier PL-programa $P$ se cumple que $P \in S \Leftrightarrow h(P) \notin S$.

Aplicando el Teorema del Punto Fijo sobre la función $h$ y el predicado $f$ llegamos a que existirá un PL-programa $E$ tal que $f(x) \in M(E) \Leftrightarrow f'(x) \in M(h(E))$.

Esto nos lleva a un absurdo, ya que al ser $S$ propiedad sobre un predicado de la semántica, tendríamos que $P \in S \Leftrightarrow h(P) \in S$, lo que contradice lo antes demostrado. □

Creemos que esta demostración es una buena manera de cerrar esta sección, ya que nos ha permitido ver una interesante aplicación del Teorema del Punto Fijo. Vemos como este teorema nos ha permitido demostrar uno de los teoremas fundamentales de la teoría de la computabilidad de una manera mucho más sencilla que la que hicimos en su momento.

Tanto con esta demostración como con el estudio de los programas autorreplicantes buscábamos dejar claro que, aunque los Teoremas de Recursión y del Punto Fijo puedan parecernos abstractos, van a ser útiles al permitirnos alcanzar otros resultados interesantes.

# Capítulo 8

# Conclusiones

Hemos desarrollado en este trabajo un modelo de computabilidad basado en programas lógicos puros. Nuestro objetivo era, una vez que hubieramos demostrado que se trata de un modelo Turing-completo, tratar de llegar a resultados clásicos de la Teoría de la Computabilidad desde esta nueva perspectiva.

Vamos a comenzar este capítulo exponiendo cuáles han sido las principales conclusiones que se han alcanzado, prestando especial atención en señalar las diferencias que hemos encontrado con otros modelos clásicos de computabilidad. Por último, expondremos los aspectos del trabajo que quedarían pendientes.

Comenzamos el trabajo estableciendo la semántica de nuestros PL-programas. Lo hicimos de una manera completamente abstracta, sin desarrollar ningún modelo de cómputo. Esta va a ser una de las principales diferencias que va a presentar nuestro modelo de computabilidad respecto a otros modelos clásicos, como las máquinas de Turing o las máquinas de registros.

Una vez establecida esta semántica pasamos a desarrollar la definición de función PL-computable. Demostramos que se trata de un modelo Turing-completo, lo que nos sirvió como motivación para el resto del trabajo, ya que nos asegura que toda función computable dentro de alguno de los modelos clásicos de computabilidad también lo va a ser para nuestros PL-programas, por lo que tiene sentido estudiarlos.

Hemos desarrollado también un PL-programa universal para nuestros programas lógicos. Para el desarrollo de este programa aprovechamos la capacidad de los PL-programas de poder expresarse como términos. Gracias a esta propiedad el desarrollo de un PL-programa universal resultó bastante sencillo y, además, permitió demostrar que este PL-programa realmente se comporta como tal. Esta es una de las principales ventajas que nos aporta el uso de los PL-programas, ya que en los modelos clásicos demostrar que sus programas universales son realmente metaintérpretes es inasumible.

Hemos desarrollado los conceptos de conjunto PL-recursivo y PL-recursivamente enumerable. Hemos demostrado que el conjunto $H_f$, definido en 5.5.1, no es PL-recursivo, lo que nos permitió demostrar encontrar ejemplos de funciones no PL-computables. También hemos utilizado reducciones para encontrar más ejemplos de conjuntos no PL-recursivos. A diferencia de en otros modelos, como las máquinas de Turing o las máqunas de registros, hemos podido realizar estas reducciones de manera directa sin tener que recurrir a resultados como el Teorema s-m-n. Al igual que en el caso del PL-programa universal, esta ventaja se debe a la capacidad de expresar PL-programas como términos de otros PL-programas.

Por último, hemos desarrollado algunos de los resultados clásicos de la Teoría de la Computabilidad. De este modo, hemos demostrado el Teorema de Rice, siendo capaces de adaptar de manera sencilla la demostración de este resultado para máquinas de Turing. También hemos podido demostrar el Teorema s-m-n. En este caso la demostración se simplifica mucho respecto a la de otros modelos de computabilidad, gracias a la representación que hemos tomado de los PL-programas como términos,

que nos permite añadir cláusulas en ellos de manera directa. Gracias a este teorema somos capaces de demostrar otros resultados como el Teorema de Recursión y el Teorema del Punto Fijo de manera sencilla.

Hemos podido comprobar en estas demostraciones que el uso de programas lógicos nos permitía razonar fácilmente aprovechando la propia estructura de las cláusulas del programa entendidas como implicaciones lógicas. Esta es una de las ventajas del carácter abstracto de la semántica que hemos tomado.

Sin embargo, hemos podido comprobar que el uso de los PL-programas también nos ha llevado a encontrarnos con ciertas dificultades respecto a otros modelos de computabilidad.

Una de las más importantes es la dificultad que presentan los programas lógicos para demostrar la PL-recursividad de ciertos conjuntos. Hemos desarrollado en este trabajo resultados de apariencia sencilla, como demostrar que el conjunto formado los naturales (desarrollado en 5.3.1) es PL-recursivo. Mientras que demostrar que este conjunto es PL-recursivamente enumerable es muy sencillo, probar que su complementario no lo es resulta más complicado de lo que en un principio podíamos esperar. Esto se debe a la propia naturaleza de los programas lógicos, muy útil para encontrar los elementos que cumplen una condición, pero que no resulta la más indicada para hallar los que no la cumplen.

Otro de los problemas que hemos encontrado, y que sería el principal trabajo que quedaría pendiente, es establecer un modelo de cómputo para nuestros PL-programas. El no tener este modelo nos ha llevado a tener algunas dificultades a la hora de desarrollar este trabajo, ya que en otros modelos clásicos de computabilidad las nociones de parar en tiempo finito o no hacerlo resultan muy importantes. Un ejemplo de esta dificultad lo encontramos a la hora de probar el Teorema de Rice-Saphiro, cuya demostración suele basarse en la idea de no parar en tiempo finito.

De este modo, la principal tarea pendiente sería desarrollar un modelo de cómputo para nuestros PL-programas. Creemos que una manera apropiada de hacerlo sería tomar un cómputo basado en la búsqueda en anchura sobre el árbol de derivación de nuestras cláusulas (conocido como árbol de SLD-resolución), como discutimos en 3.3. Necesitamos búsqueda en anchura porque la búsqueda en profundidad, adoptada por Prolog, no es completa con relación a la semántica.

El trabajo a realizar sería demostrar que la semántica obtenida a través de este modelo coincide con la que hemos tomado en este trabajo. Además, sería interesante estudiar cómo se reflejarían las nociones de PL-computable y PL-recursivo sobre esta nueva caracterización de la semántica.

# Apéndice A

# Unión de PL-programas manteniendo la independencia de sus semánticas

Antes de comenzar este anexo comenzamos definiendo $C$ como una signatura finita de constructoras cualquiera, $\mathcal{T}_C$ el conjunto de términos básicos que podemos construir a partir de ella y un conjunto de variables $V$. Trabajaremos en todo este anexo con programas construidos a partir de ellos.

Como hemos ido comentando a lo largo de este trabajo, en numerosas ocasiones vamos a querer unir PL-programas sin que los predicados de los programas se vean afectados por esta unión. Un ejemplo de esto podríamos encontrarlo en demostración de que el conjunto de las funciones PL-computables es Turing-completo, que podemos encontrar en 3.2.1. De esta manera, queremos que si $P$ es el PL-programa que une los PL-programas $P_i$, para cada predicado $f$ de uno de los programa $P_i$ tengamos que existe un predicado $f'$ tal que $f \in M(P_i) \Leftrightarrow f' \in M(P)$

Una primera idea podría ser unir nuestros PL-programas de manera directa, sin embargo esto no va a resultar conveniente porque podría alterar completamente el significado de nuestros programas en el caso de que ambos estén manejando los mismos predicados.

Para solucionar este problema, podríamos pensar que bastaría con renombrar los predicados de los programas de manera que no corramos el riesgo de que estos no coincidan. Aunque esta estrategia se acerca mucho más a la solución, no llega a ser completamente correcta debido a que estamos trabajando con signaturas de constructoras finitas. Si realizamos este renombramiento de manera directa, sustituyendo la constructoras de los programas que se repitan por otras que no se estén utilizando corremos el riesgo de agotar las constructoras de nuestro $C$.

Nos resulta sencillo comprender este problema observando el siguiente ejemplo: supongamos que $C$ tiene cardinalidad $k$, vamos a querer unir los $k+1$ programas $P_i$ de la forma siguiente:

```
f(X) :- cuerpo_i.
```

Probamos a sustituir la constructora $f$ de nuestros PL-programas $P_i$ por la constructora $f_i$. De este modo se cumpliría que $f_i(x) \in M(P) \Leftrightarrow f(x) \in M(P_i)$, como queremos. Sin embargo, estamos cometiendo un error, ya que estamos utilizando $k+1$ constructoras para representar los distintos predicados $f_i$, sin embargo tan solo tenemos en nuestra signatura de constructoras $k$, por lo que evidentemente estamos cometiendo un error.

Una posible solución sería considerar signaturas de constructoras numerables en vez de finitas, pero la condición de que $C$ sea finita es fundamental en muchas de las demostraciones desarrolladas a lo largo del trabajo. Por este motivo, hemos tratado de adaptar esta técnica de manera que no utilice más que una constructora para realizar la unión. Lo único que necesitaremos para poder hacerla es que $C$ contenga una constructora $une/2$.

**Lema A.0.1.** *Dados los PL-programas $P_0$, $P_1$, entonces podemos definir un PL-programa $P$ tal que en uno de sus predicados $une/2$ se verifique para todo término $x \in \mathcal{T}_C$*

$$une(x, cero) \in M(P) \Leftrightarrow x \in M(P_0) \text{ y } une(x, suc(cero)) \in M(P_1) \tag{A.1}$$

*Demostración.* Vamos a demostrar el resultado construyendo el PL-programa $P$ que buscamos.

Nuestros PL-programas $P_0$ y $P_1$ están formados por cláusulas $R$ de la forma:

```
R: cabeza :-
      cuerpo_1,
...
      cuerpo_n,
```

La idea que perseguimos al construir $P$ es que sea capaz de distinguir para una de estas clásulas y términos a cúal de los PL-programas $P_0$ o $P_1$ pertenece. Pues bien, para ello llamaremos $\hat{R}_0$ a la cláusula:

```
une(cabeza,cero) :-
      une(cuerpo_1, cero),
      ...
      une(cuerpo_n cero).
```

Siguiendo una notación similar, llamaremos $\hat{R}_1$ a:

```
une(cabeza,suc(cero)) :-
      une(cuerpo_1, suc(cero)),
      ...
      une(cuerpo_n suc(cero)).
```

Pues bien, tomamos como $P$ el programa que este formado por la clásula $\hat{R}_{i0}$ de cada cláusula $R_i$ de $P_0$ y $\hat{Q}_{i1}$ de cada cláusula $Q_i$ de $P_1$. Es decir, $P = \{\hat{R}_0 : R \in P_0\} cup \{\hat{R}_1 : R \in P_1\}$. Evidentemente, como $P_0$ y $P_1$ son finitos $P$ también lo será.

Demostrar que este programa se comporta como esperamos se podría hacer de manera formal, pero hemos decidido no hacerla porque consideramos que el resultado queda lo suficientemente claro por construcción. De todos modos, nos gustaría indicar que una manera sencilla de demostrarlo sería utilizando inducción sobre la forma del árbol de derivación de nuestras cláusulas.

□

Acabamos de probar que vamos a poder unir PL-programas manteniendo sus semánticas completamente independientes. A lo largo de este trabajo vamos a hacer esto constantemente, por lo que aunque siempre que hablemos de unión de PL-programas nos estemos refiriendo a este tipo de unión.

Cometeremos habitualmente abusos de notación al referirnos a predicados del programa generado por la unión por el nombre original de los predicados del antes de la unión y no el que presentarán en ella (que será de la forma $une(t, cero)$). Buscamos con esto simplificar la notación que utilizaremos en nuestras demostraciones y hacerlas más intuitivas. De todos modos, es importante tener en cuenta que realmente cuando hablemos de unión de PL-programas nos estaremos refiriendo a este tipo de unión (y no a la unión directa de los PL-programas).

Nos referiremos a lo largo del trabajo a este tipo de unión de PL-programas en que mantenemos las semánticas independientes por $\hat{\bigcup}$.

# Apéndice B

# Resultados acerca del PL-programa universal $U$

En este anexo vamos a desarrollar una serie de lemas que nos permitan demostrar el teorema 4.2.1. Lo que haremos es tratar de demostrar formalmente que todas las cláusulas que desarrollamos en el PL-programa $U$ 4.2 van a tener comportarse de la manera que buscábamos al definirlas.

Comenzamos estudiando como se comporta el objetivo *miembro* en nuestro PL-programa. Como hemos querido dejar claro al desarrollar $U$, nuestra intención es que este objetivo nos permita estudiar si un elemento pertenece a una lista. Vamos a demostrar que efectivamente nos lo va a permitir.

**Lema B.0.1.** *Dada un término básico que representa una lista cualquiera $l$ y un término $x$ se verifica que $x$ forma parte de $l$ si y solamente si $miembro(l, x) \in M(U)$.*

*Demostración.* Veamos de manera independiente cada una de las dos implicaciones. Aunque en los lemas posteriores vamos a ver como podemos demostrarlo de foma directa, hemos decidido hacer esta demostración de forma separada para que quede más claro el procedimiento seguido, que va a ser similar en muchos de los otros lemas que vamos a probar.

Tenemos por tanto, las siguientes implicaciones:

1. Si $x$ es miembro de $l$, demostraremos que $miembro(l, x) \in M(U)$. Vamos a proceder por inducción sobre la posición que ocupa el elemento $x$ sobre la lista.

   *a*) Nuestro caso base es cuando $x$ es el primer elemento de la lista, entonces $l$ es de la forma $lista(x, \dots)$. De este modo, $miembro(l, x)$ es instancia de la cabeza de la clásula

   ```
   miembro(lista(X,_),X)
   ```

   Como el cuerpo de esta cláusula es vacío, tenemos que $miembro(l, x) \in M(U)$.

   *b*) Veamos ahora el paso inductivo, es decir, cómo asumir que la propiedad se verifica hasta el $n$-esimo elemento de toda lista nos lleva a que también se verifica para el $n + 1$.

   Observamos que $l$ es de la forma $[elem, l']$ siendo $elem$ su primer elemento y $l'$ una lista donde $x$ aparece en la $n$ posición. Por la hipótesis de inducción, tenemos que $miembro(l', x) \in M(U)$.

   Ahora bien, por ser $M(P)$ modelo, llegamos a que por la cláusula

   ```
   miembro(lista(y,l'),x) :- miembro(l',x)
   ```

   tenemos que como $miembro(l', x) \in M(u)$ entonces $miembro(l, x) \in M(U)$ al ser instancia de esta cláusula.

2. Si $miembro(l, x) \in M(U)$ vamos a demostrar que $x$ es miembro de $l$. Lo haremos de manera bastante similar, utilizando tambien inducción, pero esta vez sobre la longitud de la lista.

*a*) Si es de longitud 0, no puede darse que $miembro(l, x) \in M(U)$, ya que no es instancia de ninguna de las cláusulas de $U$, por lo que tenemos el resultado.

*b*) Estudiamos ahora el caso de una longitud de lista $n + 1$, asumiendo que el resultado se verifica para toda lista de longitud $n$. Tenemos que si $miembro(l, x) \in M(U)$ entonces se ha podido llegar a esto como resultado de dos posibles clásulas:

Si se ha llegado debido a que $miembro(l, x)$ es intancia de la clásula

```
miembro(lista(X,_),X).
```

entonces tenemos que $l$ debe ser de la forma $lista(x, \dots)$, por lo que $x$ forma parte de ella.

Si se ha llegado debido a que es instancia de la clásula

```
miembro(lista(_,Resto),X) :- miembro(Resto,X).
```

tenemos que debe cumplirse que $l$ es de la forma $lista(z, l')$ y $miembro(l', x) \in M(U)$. Como $l'$ es una lista de longitud $n$, por nuestra hipótesis de inducción tenemos que $x$ es miembro de $l'$ y, por tanto, también de $l$.

□

Acabamos de probar una primera propiedad acerca de $M(U)$. Pasamos ahora a ver otras, que como vamos a ver van a demostrarse siguiendo un esquema muy similar. Este es el motivo que nos ha llevado a analizar con tanto detalle esta primera demostración.

Vamos a ver estudiar ahora el objetivo *distinto*. Como dijimos al desarrollarlo, lo queremos es que nos permita establecer cuándo dos naturales son distintos entre si, lo que usaremos para reconocer cuándo dos variables no son la misma. Este es el motivo por el que hemos decidido expresar las variables como números naturales para poder saber cuando no coinciden. Como vamos a comprobar, *distinto* nos va a permitir hacerlo.

**Lema B.0.2.** *Dados dos términos que representan naturales $m$ y $n$, expresados a través de suc y cero, se verifica que $distinto(m, n) \in M(U)$ si y solo si $m \neq n$.*

*Demostración.* Al igual que en el lema anterior, vamos a ver de manera separada cada una de las implicaciones. A partir de esta demostración trataremos de hacerlo de manera conjunta.

1. Si $distinto(m, n) \in M(U)$, veamos que $m \neq n$. Lo probaremos por inducción sobre el valor del mínimo de $m$ y $n$.

   *a*) Si este mínimo es *cero* y se trata de $m$ (para $n$ actuaríamos igual) tenemos que nuestro objetivo $distinto(cero, n)$ tan solo puede ser instancia de la cláusula

   ```
   distinto(cero, suc(_)).
   ```

   De este modo $n$ tendrá que ser de la forma $suc(n')$ con lo que es distinto de $m$.

   *b*) Si el mínimo es $k + 1$ y asumimos que hemos demostrado el resultado hasta mínimo $k$, tenemos que podemos expresar nuestros naturales como $suc(m')$ y $suc(n')$ respectivamente, cumpliéndose que el mínimo que $m'$ y $n'$ es $k$. La única clásula de la que puede ser instancia es

   ```
   distinto(suc(X),suc(Y)) :- distinto(X,Y).
   ```

   De este modo, como $distinto(suc(m'), suc(n')) \in M(U)$ llegamos a que $distinto(m', n') \in M(U)$ por ser modelo y la única clásula de la que puede ser instancia. Ahora bien, por nuestra hipótesis de inducción tenemos que entonces $n' \neq m'$, lo que nos lleva a $n \neq m$.

2. Veamos ahora la implicación contraria, si $n \neq m$ entonces $distinto(m, n) \in M(U)$. Procedemos también por inducción sobre el valor del mínimo entre $m$ y $n$.

   *a*) Si el mínimo es *cero* y se trata de $m$ (para $n$ el razonamiento es análogo) tenemos que nuestros naturales serán de la forma $m = cero$ y, como es distinto, $n$ de la forma $suc(n')$. De este modo, el objetivo $distinto(cero, suc(n'))$ es intancia de la cláusula

```
distinto(cero, suc(_)).
```

y como el cuerpo es vacío llegamos a $distinto(cero, suc(n')) \in M(U)$.

*b*) Estudiamos ahora el caso en que el mínimo es $k+1$ y hemos verificado la propiedad para los naturales hasta $k$. Podemos escribir $m$ y $n$ como $suc(m')$ y $suc(n')$ respectivamente, cumpliéndose que el mínimo entre $m'$ y $n'$ es $k$ y que $m' \neq n'$.

De este modo, por hipótesis de inducción tenemos que $distinto(m', n') \in M(U)$. Ahora bien, como $distinto(suc(m), suc(n))$ es instancia de la cláusula

```
distinto(suc(X),suc(Y)) :- distinto(X,Y).
```

y al ser $M(U)$ modelo llegamos a que $distinto(m, n) \in M(U)$.

□

Acabamos de demostrar que $U$ nos permite reconocer cuándo dos naturales son distintos entre si. Como antes mencionamos, este resultado nos interesa de cara a estudiar si una variable se había asignado más de una vez.

Lo que nosotros queremos hacer es ver si una asignación es válida, es decir, en ella no aparece ninguna variable asignada más de una vez. Para poder hacer eso, lo que haremos es tomar una a una las variables de la lista de asignaciones y ver que solo van a aparecer una vez en ella.

Para hacer esto nos resultará útil conocer cuándo una variable no aparece ninguna de las asignaciones de naturales a términos de una lista, lo que llamaremos no estar asignada. Antes de pasar a demostrar que $U$ nos va a permitir conocer esto, recordamos que en $U$ hemos utilizado la constructora $asigna(nVar, ter)$ para referirnos a la asignación de la variable $nVar$ al termino $ter$ (esto es una cuestión meramente sintáctica).

Como veremos, se va a tratar de una demostración muy sencilla y similar a las que venimos haciendo, en la que usaremos de nuevo inducción matemática. Como sigue el mismo esquema que las anteriores sin presentar ningun dificultad añadida vamos a hacerla menos detenidamente.

**Lema B.0.3.** *Dado un término que representa una lista de asignaciones de naturales a términos $l$ y un término que representa un natural $x$ se verifica que $x$ no está asignado en $l \Leftrightarrow noPertenece(x, l) \in M(U)$.*

*Demostración.* Procedemos por inducción sobre la longitud de la lista $l$.

1. Para longitud 0 evidentemente tenemos que $l = vacia$. De este modo, tenemos que el objetivo que estamos estudiando es $noPertenece(x, vacia)$ que es instancia de la cláusula

   ```
   noPertenece(_,vacia).
   ```

   De este modo, tenemos que $noPertenece(x, l) \in M(U)$.

   Evidentemente, también se comple que $x$ no está asignado en $l$, por lo que se cumple nuestro enunciado.

2. Estudiamos el caso de una lista $l$ de longitud $k+1$ asumiendo que el enunciado se verifica para listas de longitud $k$.

   Tenemos que la lista $l$ es de la forma $lista(asigna(y, t), l')$ siendo $l'$ una lista de asignaciones de naturales a términos de longitud $k$. Entonces como $x$ no estará asignado en $l$ si y solo si $x \neq y$ y $x$ no está asignado en $l'$. Mediante la hipótesis de inducción y el lema anterior,vemos que esto sucederá si y solamente si $distinto(x, y)$ y $noPertenece(x, l')$ pertenecen a $M(U)$.

   Finalmente, como

```
noPertenece(X,lista(asigna(Y,_),Resto)) :-
    distinto(X,Y),
    noPertenece(X,Resto).
```

es la única cláusula de $U$ de la que el objetivo $noPertenece(x, l)$ puede ser instancia, entonces se cumple que $distinto(x, y), noPertenece(x, l') \in M(U) \Leftrightarrow noPertenece(x, l) \in M(U)$. Combinando esto con lo que antes teníamos llegamos a que $x$ no está asignado en $l \Leftrightarrow noPertenece(x, l) \in M(U)$.

□

Ahora que tenemos este resultado, ya vamos a poder estudiar lo que antes nos habíamos propuesto: si una asignación de variables es válida. Esto ocurre cuando una misma variable no aparece asignada a más de un término.

Como antes habíamos expuesto, estamos representando la asignación de las variables como una lista de asignaciones individuales de naturales a términos, es decir, como $lista(asigna(nVar, t), ...)$. Lo que queremos por tanto es ser capaces de reconocer cuándo esta asignación es válida, que será cuando ningún natural aparezca repetida en ella.

Vamos a demostrar ahora cómo $U$ nos va a permitir hacerlo, siguiendo una demostración muy parecida a la que acabamos de hacer.

**Lema B.0.4.** *Dado un término básico $l$ que representa una lista de asignaciones de naturales a términos, se verifica que la asignación representada por $l$ es válida si y solo si $asigValida(l) \in M(U)$.*

*Demostración.* Vamos a demostrarlo por inducción sobre la longitud de la lista. Estudiamos los distintos casos:

1. Para listas vacías tenemos que $asigValida(vacia)$ es instancia de la cláusula

   ```
   asigValida(vacia).
   ```

   cuyo cuerpo es vacío. De este modo, llegamos a que $asigValida(vacia) \in M(U)$.

   Por otro lado, resulta trivial que toda asignación vacía es válida.

2. Estudiamos ahora el caso de listas de longitud $k+1$ asumiendo que se verifica el enunciado para las de longitud $k$. Podemos escribir la lista $l$ como $lista(asigna(x, t), l')$ siendo $l'$ de longitud $k$ y $asigna(x, t)$ la primera asignación de la lista.

   Tendremos que la asignación $l$ será válida si y solo si $x$ no aparece en $l'$ (en caso contrario estaría repetida en $l$) y además la asignación de $l'$ es válida. Por hipótesis de inducción tenemos que $l'$ es válida $\Leftrightarrow asigValida(l') \in M(U)$ y por el lema B.0.3 $x$ no pertenece a $l' \Leftrightarrow noPertenece(x, l') \in M(U)$.

   De este modo, llegamos aplicando estos resultados nuestra definición de asignación válida a que $l$ es válida $\Leftrightarrow asigValida(l'), noPertenece(x, l') \in M(U)$.

   Ahora bien, como la cláusula

   ```
   asigValida(lista(asigna(X,_),Resto)) :-
         noAsignado(X,Resto),
         asigValida(Resto).
   ```

   es la única de $U$ de cuya cabeza $asigValida(l)$ puede ser instancia, se cumple que $l$ es válida $\Leftrightarrow noAsignado(x, l'), asigValida(l') \in M(U) \Leftrightarrow asigValida(l) \in M(U)$.

□

Pasamos ahora a uno de los resultados principales que vamos a necesitar de cara a probar la corrección de $U$. Se trata de demostrar que $U$ nos va a permitir reconocer cuando dos términos son iguales.

Como hemos comentado, los términos de nuestros PL-programas pueden ser de dos tipos: variables o construcciones.

En ambos casos vamos a encontrar problemas de cara a ver si un término coincide con otro. Por un lado, las variables dependen de la asignación que se haya hecho de ellas. Diremos que una variable es igual a un término cuando se haya asignado la variable a dicho término. Por otro, para las construcciones tendremos que comprobar tanto que el nombre del constructor coincide como que cada uno de los argumentos son iguales.

Aunque en un principio puede parecer complicado probar que el objetivo *comparaTermino* va a comportarse de este modo, vamos a ver cómo, al igual que en los casos anteriores, vamos a ser capaces de alcanzar el resultado mediante inducción.,

**Lema B.0.5.** *Dado un término básico que representa una asignación asig y dos términos $x$ e $y$, siendo $y$ una construcción, tenemos que $x$ e $y$ son iguales bajo la asignación asig si y solo si $comparaTermino(x, y, asig) \in M(U)$.*

*Demostración.* Distinguimos dos casos según el término $x$ sea una variable o no.

1. Si es una variable, entonces tenemos que nuestro primer término $x$ es de la forma $var(x')$. Para que coincidan necesitamos que $x'$ esté asignado en $asig$ a $y$. Esto ocurrirá si y solo si $asigna(x', y)$ pertenece a la lista $asig$. Por el lema B.0.1 una condición necesaria y suficiente para ello es que $miembro(asig, asigna(x', y)) \in M(U)$.

   Para terminar con este caso, observamos que la única cláusula de la que puede ser instancia $comparaTermino(x, y, a)$ es

   ```
   comparaTermino(var(X),Y,Asig) : - miembro(Asig,asigna(X,Y))
   ```

   por lo que podemos concluir que $miembro(asig, asigna(x', y)) \in M(U) \Leftrightarrow comparaTermino(var(x'), y, asig) \in M(U)$, lo que nos lleva al resultado enunciado.

2. Si se trata de una construcción, entonces $x$ será de la forma $const(nombre, args)$.

   Aprovechando que sabemos que $y$ no es una variable, tenemos que también será una construcción $y = const(nombre', args')$. Como antes hemos expuesto, para que $x$ e $y$ coincidan necesitamos que $nombre$ coincida con $nombre'$ y que la lista $args$ sea igual a $args'$ término a término dada la asignación $asig$.

   Para demostrar el resultado vamos a proceder por inducción estructural sobre $x$. Tenemos que $arg$ es una lista de términos, que es la que nos va a permitir realizar este tipo de inducción. De este modo, suponemos que para cada uno de los términos que constituyen $arg$ se verifica el enunciado, demostraremos que en tal caso también se cumple para $x$.

   Antes de pasar a la inducción, observamos que la única cláusula de la que podría ser instancia nuestro objetivo $comparaTermino(x, y, asig)$ es

   ```
   comparaTermino(const(X,Y),const(X,Y1),Asig) :- comparaLista(Y,Y1,Asig).
   ```

   De este modo, tenemos que $comparaTermino(x, y, asig) \in M(U) \Leftrightarrow nombre = nombre'$ y $comparaLista(arg, arg', asig) \in M(U)$.

Lo que nos queda por demostrar es que $arg$ coincide con $arg'$ si y solo si $comparaLista(arg, arg', asig) \in M(U)$. Aplicamos para probarlo inducción sobre la longitud de la lista $arg$, además de la inducción estructural que ya teníamos.

*a*) Si $arg$ es vacía, entonces tenemos que la única cláusula de la que el objetivo $comparaLista(arg, arg', asig)$ podría ser instancia es

```
comparaLista(vacia,vacia,_).
```

Para ser instancia de ella únicamente necesita es que $arg' = arg$, lo que nos lleva a que $arg = arg' \Leftrightarrow comparaLista(arg, arg', asig) \in M(U)$.

*b*) Probemos ahora que se va a verificar para listas de longitud $k+1$ asumiendo que la propiedad se cumple en las de longitud $k$.

Como $arg$ es de longitud $k+1$, entonces será de la forma $lista(x', l)$ siendo $x$ un término y $l$ lista de términos de longitud $k$. Por lo tanto, para que $arg$ coincida con $arg'$ término a término necesitaremos que $arg'$ sea de la forma $lista(y', l')$ cumpliéndose que el término $x'$ coincide con $y'$ y la lista $l$ con $l'$ para la asignación $asig$. Por la hipótesis de inducción estructural y de la longitud de $arg$ respectivamente, tenemos que $x'$ e $y'$ coincidirán si y solo si $comparaTermino(x', y', asig) \in M(U)$ y $l$ y $l'$ si y solo si $comparaLista(l, l', asig) \in M(U)$.

Terminamos la demostración viendo que nuestro objetivo $comparaLista(arg, arg', asig)$ tan solo puede ser instancia de la cláusula:

```
comparaLista(lista(X,Resto),lista(Y,Resto1),Asig) :-
    comparaTermino(X,Y,Asig),
    comparaLista(Resto,Resto1,Asig).
```

Esto nos lleva finalmente a que $arg$ coincide con $arg' \Leftrightarrow comparaTermino(x', y', asig)$, $comparaLista(l, l', asig) \in M(U) \Leftrightarrow comparaTermino(x, y, asig) \in M(U)$.

Volvemos ahora al punto de nuestra demostración del enunciado en que nos encontrábamos. Teníamos que $comparaTermino(x, y, asig) \in M(U) \Leftrightarrow nombre = nombre'$ y se cumple que $comparaLista(arg, arg', asig) \in M(U)$.

Ahora bien, como acabamos de ver que esto sucede si y solo si los constructores $nombre$ y $nombre'$ coinciden y $arg$ y $arg'$ también lo hacen.

Por lo tanto, hemos alcanzado que $comparaTermino(x, y, asig) \in M(U) \Leftrightarrow x$ e $y$ son iguales.

Como hemos visto que en ambos casos llegamos al resultado deseado, podemos acabar la demostración. □

Acabamos por tanto de probar que el objetivo *comparaTermino* nos permite reconocer cuando dos términos coinciden. Esto va a ser fundamental para nosotros, ya que lo usaremos para estudiar cuando una cláusula básica es instancia de una del programa. Para que lo sea, es necesario que exista una asignación de variables para la cual las dos cláusulas sean iguales, es decir, que tanto sus cabezas (que aunque se trate de un objetivo tiene la estructura de un término) como sus cuerpos coincidan.

Pasamos ahora a demostrar que también vamos a poder reconocer cuándo dos listas de términos son iguales en el sentido de que los términos de ambas coinciden uno a uno. Este resultado ya lo hemos probado de manera indirecta en el anterior lema, por lo que su demostración va a ser trivial.

**Lema B.0.6.** *Dados dos términos que representan listas de términos arg y arg' y un término que representa una asignación de variables a, se verifica que arg y arg' coinciden bajo la asignación a si y solo si* $comparaLista(arg, arg', a) \in M(U)$.

*Demostración.* Ya hemos demostrado este resultado de modo indirecto en la demostración del lema anterior.

Hemos visto cómo si se asumía la hipótesis (ahora ya demostrada) de que para los términos de $arg$ se cumplía B.0.5 entonces teníamos que las listas $arg$ y $arg'$ iban a coincidir dada la asignación $a$ si y solo si $comparaLista(arg, arg', a) \in M(U)$.

Dado que hemos probado que esto siempre es cierto, entonces $arg$ y $arg'$ coinciden dada una asignación $a$ si y solo si $comparaLista(arg, arg', a) \in M(U)$ □

Vemos cómo estamos siendo capaces de demostrar que $U$ va a computar cada vez objetivos más complejos. Hemos pasado de sencillos objetivos como que dos naturales sean distintos a resultados cada vez más complejos, como la igualdad de términos.
Vamos a ir ahora un paso más allá probando que $U$ va a computar un objetivo que nos permite reconocer cuándo una cláusula básica puede ser instancia de otra cláusula.

Queremos resaltar cómo este resultado va a ser sencillo de probar gracias a todos los lemas intermedios que hemos ido desarrollando en este anexo.

**Lema B.0.7.** *Sean términos $r_1$ y $r_2$ que representan dos cláusulas tales que $r_2$ es básica. Entonces se verifica que $r_2$ es instancia de $r_1$ si y solo si $esInstancia(r_1, r_2) \in M(U)$.*

*Demostración.* La demostración va a resultarnos muy sencilla gracias a los resultados que hemos ido demostrando.

Tenemos que $r_2$ será instancia de $r_1$ cuando exista una asignación válida de las variables de $r_1$ de tal manera que tanto la cabeza (que es un objetivo) como el cuerpo (lista de objetivos) de $r_1$ y $r_2$ coincidan. Escribiremos la asignación $asig$ como una lista de la forma $lista(asigna(cero, t), lista(\dots))$ como hemos venido haciendo, con el objetivo de poder utilizar los lemas que hemos demostrado.

Observamos ahora que la sintaxis de $r_1$ y $r_2$ se corresponde respectivamente con $regla(ter_1, lista_1)$ y $regla(ter_2, lista_2)$. De este modo, para que $r_1$ y $r_2$ coincidan bajo la asignación válida de variables $asig$ lo que deberá ocurrir es que $ter_1$ coincida con $ter_2$ y $lista_1$ con $lista_2$, en ambos casos con respecto a la asignación de $asig$.

Como hemos probado en los lemas B.0.5 y B.0.4, esto ocurre si y solamente si existe una asignación $asig$ tal que $asigValida(asig)$, $comparaTermino(ter_1, ter_2, asig)$, $comparaLista(lista_1, lista_2, asig) \in M(U)$.

Finalmente, como la cláusula

```
esInstancia(regla(Cabeza,Cuerpo),regla(Cabeza1,Cuerpo1)) :-
    comparaTermino(Cabeza,Cabeza1,Asig),
    comparaLista(Cuerpo,Cuerpo1,Asig),
    asigValida(Asig).
```

es la única con el objetivo *esInstancia* en su cabeza, tenemos que existirá una asignación $asig$ tal que se cumpla que $asigValida(asig)$, $comparaTermino(ter_1, ter_2, asig)$, $comparaLista(lista_1, lista_2, asig) \in M(U)$ si y solo si $esInstancia(r_1, r_2) \in M(U)$.

□

Vamos a pasar ahora al último resultado intermedio que necesitamos de cara a probar que $U$ es efectivamente un PL-programa universal. Demostraremos que $U$ permite reconocer cúando una cláusula básica es instancia de una de las cláusulas de un programa $P$.

Para ello evidentemente va a ser de gran ayuda el resultado que acabamos de demostrar. Sabemos que la cláusula ground $r_2$ es instancia de $r_1$ si y solo si $esInstancia(r_1, r_2) \in M(U)$. Así que si queremos saber si $r_2$ puede ser instancia de alguna cláusula de $P$ nos bastará con comprobar si alguna de

las cláusulas $r_P$ que conforman $P$ cumple el objetivo $esInstancia(r_P, r_2)$.

Veamos ahora como podemos fomarlizar esta idea de cara a demostrar el resultado siguiente, aunque antes nos gustaría hacer notar que estamos haciendo un abuso de notación al referirnos por $P$ tanto al PL-programa como a su representación como término.

**Lema B.0.8.** *Dado obj objetivo ground, una lista lista de objetivos ground y un PL-programa $P$ se tiene que la cláusula $regla(obj, lista)$ es instancia de alguna cláusula de $P$ si y solamente si $existeRegla(obj, P, lista) \in M(U)$.*

*Demostración.* La demostración al igual que en el caso anterior va a ser muy sencilla y nos va a permitir llegar al resultado sin apenas dificultades gracias a los lemas que hemos desarrollado en esta sección.

Suponemos que $regla(obj, lista)$ es instancia de alguna cláusula de $P$. De este modo existe una cláusula $r$ en $prog$ (que lo hemos definido como una lista de cláusulas) tal que $regla(obj, lista)$ es instancia suya. Aplicando los lemas B.0.1 y B.0.7 tenemos que esto sucederá cuando exista una cláusula $r$ tal que $miembro(prog, x), esInstancia(r, regla(obj, lista)) \in M(U)$.

Por último, como la cláusula

```
existeRegla(A,P,Cuerpo) :-
miembro(P,R),
esInstancia(R,regla(A,Cuerpo)).
```

es la única con $existeRegla$ en su cabeza, con lo que es la única de la que $existeRegla(obj, prog, lista)$ puede ser instancia. Por lo tanto, que existe tal $r$ cumpliendo $miembro(l, x)$, $esInstancia(r, regla(obj, lista)) \in M(U)$ si y solo si $existeRegla(obj, P, lista) \in M(U)$.

□

Damos con la demostración de este lema por finalizado este anexo. En él hemos reunido una serie de lemas intermedios que nos van a permitir demostrar el teorema 4,2,1, que es uno de los resultados más importantes de este trabajo ya que es la base de todo el desarrollo de la PL-computabilidad que realizaremos más adelante.

Nos gustaría señalar como no hemos tenido que utilizar ninguna técnica compleja de cara a desarrollar estos lemas. Nos hemos limitado a utilizar distintos tipos de inducción y aprovechar los resultados que íbamos desarrollando, permitiéndonos de este modo alcanzar resultados interesantes acerca de $U$ sin tener que realizar complicados desarrollos. Esta es una de las principales ventajas que nos ofrece el uso de programas lógicos, ya que al trabajar con semánticas tan abstractas nos va a resultar sencillo razonar sobre ellas.

# Apéndice C

# PL-programa que reconoce si un término representa un PL-programa

Vamos a desarrollar en este anexo un PL-programa que sea capaz de reconocer cuándo un término representa a un PL-programa, siguiendo la sintaxis para referirnos a programas como términos que tomamos en 4.1.

Comenzamos tomando $C$ una signatura de constructoras finita de la forma $C = \{c_1/n_1, \ldots, c_k/n_k\}$.

Lo que buscamos de este modo es un PL-programa, al que denotaremos por $V$, tal que para todo término $x \in \mathcal{T}_C$ se cumpla que

$$esPrograma(x) \in M(V) \Leftrightarrow x \text{ representa un PL-programa construido a partir de } C \tag{C.1}$$

Tomamos $V$ el PL-programa siguiente:

```
esPrograma(vacia).
esProgramalista(R,P)) :-
      esRegla(R),
      esPrograma(P).

esRegla(regla(T, L)) :-
      esTermino(T),
      esListaTerminos(L,_).

esTermino(const(c1, L)) :- esListaTerminos(L, n1).
...
esTermino(const(ck, L)) :- esListaTerminos(L, nk).

esTermino(var(X)) :- esNatural(X).

esNatural(cero).
esNatural(suc(X)) :- esNatural(X).
```

Este programa verificará la condición que hemos establecido antes, por lo que se corresponde con el que buscamos. Es sencillo observar que así es por construcción, basándonos en las reflexiones que hicimos al establecer la sintaxis de nuestros PL-programas como términos en 4.1. Lo que estamos haciendo en este programa es tratar de adaptar las condiciones que establecimos en dicha sintaxis en forma de reglas.

Nos gustaría señalar que esto ha sido posible debido a que la signatura de constructoras con la que estamos trabajando es finita. En caso de no serlo estaríamos generando un programa con un número infinito de reglas, lo que no se corresponde con la forma de nuestros programas lógicos.

# Bibliografía


Alexander Shen, N. N. M., Nikolai Konstantinovich Vereshchagin. *Computable Functions*. American Mathematical Society, 2003.

Bridges, D. S. *Computability: A Mathematical Sketchbook*. Springer, 1994.

Church, A. *An unsolvable problem of elementary number theory*. American journal of mathematics, vol.58, 1936.

Cutland, N. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

Fernandez, M. *Models of computation*. Springer, 2009.

George S. Boolos, R. C. J., John P. Burgess. *Computability and Logic*. Cambridge University Press, 2002.

Hanne Riis Nielson, F. N. *Semantics with Applications: An Appetizer*. MIT Press, 2007.

Kleene, S. C. *General recursive functions of natural numbers*. Mathematische Annalen, 112, 1936.

Kleene, S. C. *Introduction to Metamathematics*. Ishi Press, 2009.

Leon S. Sterling, E. Y. S. *The Art of Prolog*. Springer, 1994.

L.Hein, J. *Prolog Experiments in Discrete Mathematics, Logic and Computability*. Portland State University, 2005.

Sipser, M. *Introduction to the Theory of Computation*. Wadsworth Publishing Co Inc, 2012.

Turing, A. *On computable numbers, with an application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, 2 s. vol. 42, 1937.